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

Junsung Kim, Ragunathan (Raj) Rajkumar
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA, USA
{junsungk, raj}@ece.cmu.edu

Markus Jochim
General Motors R&D
30500 Mound Rd., Warren, MI, USA
markus.jochim@gm.com

Abstract

We propose a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features such as Hot Standby and Cold Standby in order 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, SAFER reconfigures the system to recover failed processors and tasks. SAFER has been implemented on Ubuntu 10.04 LTS and deployed on Boss, an award-winning driverless vehicle developed at CMU. We provide preliminary measurements using one of the autonomous driving simulation scenarios used during the 2007 DARPA Urban Challenge.

1. Introduction

Advances in distributed embedded real-time systems have enabled a variety of different applications such as sensor networks, 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 can be of high importance for some 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.

The conventional way of improving reliability has been adding redundant hardware. However, this approach becomes less attractive to many applications because the amount of necessary hardware multiplies as the size of the system increases. It is also not consistent with the growing needs of flexible system design. Therefore, we propose a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features such as Hot Standby and Cold Standby in order to tolerate *fail-stop* processor failures and task failures for distributed embedded real-time systems in a timely manner. To detect such failures, SAFER monitors the health status and state information of each task and broadcasts the information. When a failure is detected, SAFER reconfigures the system to recover failed processors and tasks using task-level replication techniques.

SAFER targets multiple goals. Most importantly, *no single point of failure* is permitted. In other words, a task/processor failure should not lead to system failure. *Failure recovery within a guaranteed duration* should also be achieved. Embedded systems are usually tightly connected to the physical world. In such a case, failure recovery without predictable timing behavior could return unpredictable results in the physical world.

Apart from the two goals above, *predictive fault discovery and notification, resource isolation, ease of use of abstraction, ease of application development, and sensor/actuator control* are other factors considered in the design of SAFER.

The rest of this paper is organized as follows. Section 2 describes the architecture of SAFER and its implementation. Section 3 presents preliminary results measured on Boss, an award-winning autonomous vehicle developed at CMU. Section 4 presents the related work, and we conclude in Section 5.

2. The SAFER Architecture

The SAFER layer is composed of SAFER daemons (one on each processor) and a library offering a task execution environment. The library enables any task launched on the SAFER layer to be periodically executed (with reconfigurable parameters). The daemons have a master-slave architecture [2], and the master SAFER daemon controls the slave SAFER daemons responsible for managing tasks on each node[1] and monitoring its health status. The reconfigurable parameters for each task are given to the task library when the task is launched by a SAFER daemon. For the underlying communication layer, an inter-process communication primitive such as IPC [11] and SimpleComms [12] can be used. The overall architecture of the SAFER layer is illustrated in Figure 1.

The SAFER layer utilizes two main features to avoid system failure in the presence of fail-stop processor or task failures. The SAFER layer supports task-level replication techniques such as Hot Standby and Cold Standby, where

[1]In this paper, node is interchangeably used with processor.

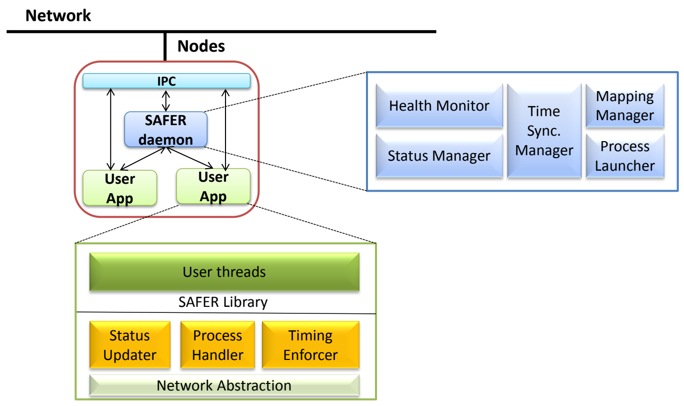

Figure 1. The overall architecture of SAFER

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 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 more critical tasks by adding more Hot Standbys/Cold Standbys for those critical tasks. We can also efficiently manage the available computing resources by not replicating less-critical tasks, thus enabling an affordable solution.

The second feature is the fail-over of the master SAFER daemon. Since the SAFER daemons have a master-slave relationship and manage tasks on each machine, the master SAFER daemon becomes a single point of failure. Hence, when the master SAFER daemon fails, one of the slave SAFER daemons will be promoted to become the master SAFER daemon. This can be done using a group membership protocol [4, 10]. Assuming a synchronous communication network[2] [4], the membership protocol of SAFER is different from the existing work in the sense that (i) we provide predictable timing behavior and (ii) the recovery duration of the SAFER daemons is deterministic. In our membership protocol, all SAFER daemons including the master and the slaves broadcast messages to each other. The master can detect the failures of a slave by the lack of heartbeat messages. The master SAFER daemon will command as necessary the slave SAFER daemons on live processors to recover any failed tasks. The death of the master can also be detected by the slaves due to the absence of heartbeats, and one of the slave SAFER daemon will be promoted to become the master by following a predetermined sequence of the slave SAFER daemons.

2.1. The SAFER Daemon

As illustrated in Figure 1, a SAFER daemon is composed of a health monitor, status manager, time synchronization manager, mapping manager and process launcher. The health monitor and status manager are responsible for monitoring the health status of the other processors and for changing the local node's role between the master and the slave. The time synchronization module offers a global time service. The mapping manager and process launcher can automatically deploy tasks on the nodes running a SAFER daemon.

2.1.1. Health Monitor and Status Manager

The health monitor periodically sends heartbeat signals to the other nodes in the system and monitors the health status of the other daemons and their processors. Therefore, the health monitor enables the SAFER daemons to agree upon the availability of each node. The period of heartbeat signals is configurable, and the list of current running tasks is added to the heartbeat signal and is broadcast to the other nodes. The status manager watches the current status of tasks running on its own node and notifies the failure of any task if there is a task failure (say due to a segmentation fault) by capturing the OS signal.

2.1.2. Time Synchronization Manager

The SAFER layer offers a global time service using a service similar to NTP [8] 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 the failure recovery occurs within the given timing requirement using a proper offset between the primary task and its Hot/Cold Standbys. This also enables the timing enforcer of the SAFER library to have less penalty in resource scheduling.

2.1.3. 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 a given system configuration file. The system configuration file includes information about where tasks are allocated and how many resources tasks demand. 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 the information from the system configuration file and updates whenever the information changes. Changes to this information can occur due to processor failures, 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 the different processors. The process mapping manager and launcher can be connected to a user-interface application such that the application 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 [7], the operator interface of Boss [13].

[2]The failure model of network is beyond the scope of this paper. We assume any packet eventually arrives at the destination.

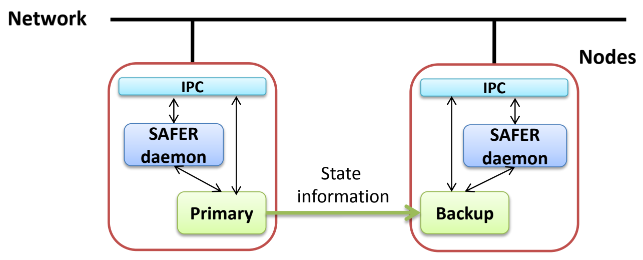

Figure 2. The primary-backup architecture

2.2. The SAFER Library

The SAFER library is a task execution environment composed of a status updater, process handler, timing enforcer and network abstraction. The user threads developed by application developers will run on the SAFER library.

2.2.1. Status Updater

The status updater of the SAFER library supports task-level replication techniques by managing state information between the primary and its Hot/Cold Standbys. The role of status updater changes based on whether a task it monitors is a primary or a backup. The backups subscribe to the primary, and the primary publishes its status. They use the Publish/Subscribe model. The status updater of the primary task periodically transfers its internal state information to its Hot/Cold Standbys, where the update period is configurable. The status updater at Cold Standby updates the state information coming from the primary in case the primary fails. The status updater can also be used as heartbeat signals by Hot/Cold Standbys. This could be useful when the period of the state updater is shorter than the update period of the SAFER daemon. This architecture is also depicted in Figure 2.

2.2.2. 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. When a backup is promoted, the new primary starts generating outputs for use and confirms its promotion to the master SAFER daemon. It must be noted that a Hot Standby is always running and its outputs are filtered by the network abstraction under the control of the process handler.

2.2.3. Timing Enforcer

The timing enforcer of the SAFER library enables tasks to have guaranteed and protected access to required processing resources in a timely manner based on Linux/RK [9]. 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. For the SAFER library, this CPU reservation model in Linux/RK is utilized.

2.3. Failure Detection and Recovery

Heartbeat signals from the health monitor of each SAFER daemon will be used for detecting processor failures. Since we have assumed a synchronous network, we have defined a time delay d to represent the maximum network delay of the heartbeat signal packets. The master SAFER daemon will decide the death of a processor unless it hears a heartbeat signal from a processor within $d + T_{heartbeat}$[3], where $T_{heartbeat}$ is the interval between two consecutive heartbeat signals. We call this failure detection scheme as *time-based 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, the mapping manager of the master SAFER daemon will be notified by the status manager, and an appropriate recovery will be initiated. We name this failure detection scheme as *event-driven 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 task-level replication techniques such as Hot Standby and Cold Standby. A Hot Standby of a task is a replicated task running concurrently with its primary. With no failure, a Hot Standby receives the same input as the primary, and the user threads of the Hot Standby do what they are supposed to do except that the outputs from them are filtered by the network abstraction[4]. In the presence of any task failure detected by the master SAFER daemon, the daemon will send a command to the SAFER daemons with the Hot Standbys for the failed tasks. Then, the process handler of each Hot Standby will receive the command to promote itself to be the primary. A similar process is also applicable to the recovery operation for Cold Standbys. One prime difference is that task needs to be launched first.

The Cold Standby of a task is a dormant binary in memory, triggered only by the failure of its primary. Without a failure, a Cold Standby periodically receives and stores the 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. We are extending Linux/RK to minimize the latency of recovery.

[3]Increasing the decision boundary can be one way of extending the assumption beyond the synchronous network. For example, many wireless networks try to send a packet n times in order to transfer it reliably, where n is a positive integer greater than 1. Then, the decision boundary can be adjusted to $d + nT_{heartbeat}$.

[4]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 should be modified to compare the results of the primary with the results of its Hot Standbys.

Task	Period	Standby	Detection	Recovery
BehaviorTask	$10ms$	Cold	$22ms$	$12ms$
ControllerTask	$10ms$	Hot	$27ms$	$9ms$
LocalPlannerTask	$100ms$	Cold	$23ms$	$66ms$
Planner3DTask_first	$100ms$	Hot	$14ms$	$28ms$
Planner3DTask_second	$100ms$	Hot	$23ms$	$66ms$

Table 1. Evaluation on time-based detection

Task	Period	Standby	Detection	Recovery
BehaviorTask	$10ms$	Cold	$2ms$	$9ms$
ControllerTask	$10ms$	Hot	$4ms$	$2ms$
LocalPlannerTask	$100ms$	Cold	$6ms$	$12ms$
Planner3DTask_first	$100ms$	Hot	$3ms$	$42ms$
Planner3DTask_second	$100ms$	Hot	$4ms$	$92ms$

Table 2. Evaluation on event-driven detection

3. Preliminary Evaluation

SAFER is implemented on Ubuntu 10.04.3 LTS and deployed on Boss which won 2007 DARPA Urban Challenge [13]. To measure the preliminary 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 fault detection time and fault recovery time for different tasks with different periods. The fault detection time is the time duration between when a failure happens and when the master SAFER daemon detects the failure. The fault recovery time is the time duration between when the master SAFER daemon detects the failure and when the failed task is completely recovered.

Table 1 captures one-time measurements when time-based detection is used. From the data, it is seen that the failure detection time depends on the period of the master SAFER daemon, which is $10ms$. All latencies are longer than $10ms$ due to d, the network time delay. 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 master SAFER daemon. Table 2 shows the measurements when event-driven detection is used. Since the local SAFER daemon detects local task failure, the failure detection time is reduced.

4. Related Work

Fault-tolerant distributed embedded systems have been extensively studied in [4, 10, 5, 1]. One clear distinction between the existing work and SAFER is that SAFER provides the framework to support timely failure recovery in a generalized setting. Fault-tolerant scheduling in distributed embedded real-time systems has also been widely researched in [3, 6], which can be potentially used for the inputs to SAFER as a configuration file.

5. 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 an autonomous vehicle developed at CMU. We have presented initial measurements using one of the driving simulation scenarios used during the DARPA Urban Challenge. Future work to be done includes supporting the graceful degradation based on load and resource changes. A comprehensive scheduling framework of the primary and its backups can also be integrated into SAFER.

References

[1] J. Balasubramanian, et al. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

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

[3] J. Chen, C. Yang, T. Kuo, and S. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2007.

[4] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

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

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

[7] M. McNaughton, C. Baker , T. Galatali, B. Salesky, C. Urmson, and J. Ziglar. Software infrastructure for an autonomous ground vehicle. *Journal of Aerospace Computing, Information, and Communication*, 5(1):491 – 505, December 2008.

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

[9] S. Oikawa and R. Rajkumar. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the fifth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1999.

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

[11] R. Simmons. Inter Process Communication (IPC). `http://www.cs.cmu.edu/~ipc` as of January 29, 2012.

[12] C. Urmson. SimpleComms. `http://www.cs.cmu.edu/~curmson/SimpleComms.tgz` as of January 29, 2012.

[13] 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.