Feature Article:

DOI. No. 10.1109/MAES.2019.2905921

Enabling Strong Isolation for Distributed Real-Time Applications in Edge Computing Scenarios

Abhishek Dubey, William Emfinger, Aniruddha Gokhale, Pranav Kumar, Dan McDermet, Ted Bapty, Gabor Karsai, **Institute for Software-Integrated Systems, Vanderbilt University, Nashville, USA**

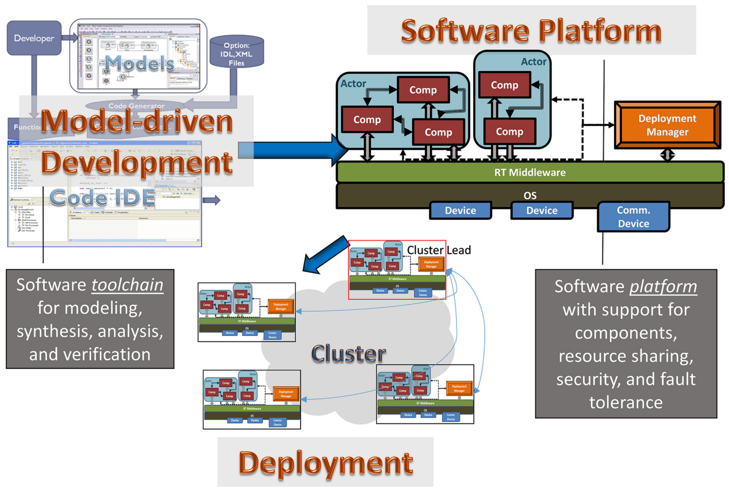
INTRODUCTION

With the advent of the cloud-computing paradigm, systems requiring intensive computations over large volumes of data have relied on the usage of shared data centers to which they transfer their data for processing. This paradigm provides a powerful scheme for applications benefiting from deep processing and data availability. It also gives rise to nontrivial problems, however, to meet the requirements of time-sensitive applications that require processing close to the source of data. While not always hard real-time in the strict sense, such applications are strongly coupled to the critical processes in the physical system and must be responsive, i.e., they must be low latency. Examples of such applications include a) wind turbine farm, where variations in wind patterns creates both control challenges (such as determining the speed and power to generate from each area) and system management challenges (such as predictive health management and prognosticating the remaining time to maintenance), b) transportation networks, where a set of cameras are used to capture video frames, which are then analyzed in real time to estimate the traffic densities, which can then be used to control the timing of the traffic lights, and c) CubeSats, where real-time data are collected and analyzed to predict the remaining useful life of battery and identify if anomalies are occurring [1]. In all these cases the delay incurred by data propagation across the backhaul communication network is not suited to serve the needs of applications we focus on, which require (near) real-time response or high Quality of Service (QoS) guarantees. The backhaul data handling latency can become severe in the unpredictable occasions where the network throughput is limited.

Authors' current addresses: A. Dubey, W. Emfinger, A. Gokhale, P. Kumar, D. McDermet, T. Bapty, and G. Karsai, are with the Institute for Software-Integrated Systems Vanderbilt University, Nashville, TN 37235, USA. E-mail: (abhishek.dubey@vanderbilt.edu).

Manuscript received September 1, 2018, revised December 31, 2018; accepted March 14, 2019, and ready for publication March 26, 2019.
Review handled by M. D. R-Moreno.
0885-8985/19/$26.00 © 2019 IEEE

Consequently, to address these needs, the research community has embraced a number of paradigms such as edge computing, cloudlets, fog computing, dispersed computing, mist computing, etc., [2]. At the core of all these paradigms is the idea that is similar to cloud computing: the sharing of computing and communication resources among colocated, multitenant applications. However, most commercial and research solutions for these technologies either choose container-based virtualization (e.g., docker) [3] or virtual machine-based virtualization [4], both of which do not necessarily provide the levels of strong isolation between critical applications that we desire in the military and space computing domains. Strong isolation in our context includes all of the following: 1) *non-bypassable,* mandatory access control on all shared (or shareable) resources; 2) temporal isolation among time-sensitive applications; 3) control over potential side-channels to prevent information leakage; and 4) fault containment applied to applications. Such isolation is very critical if the system contains one or more mixed-criticality applications, potentially sourced from different vendors, to handle multiple complex activities [5]. These applications, containing multiple processes of different criticality, require strict isolation with respect to resource guarantees, faults, and security [6]. Such requirements imply that an application's performance, the faults it encounters, or life cycle changes it goes through should not in any way impact an application residing in another isolation group [7]. Such critical applications are found particularly in military and space applications, including satellite clusters as demonstrated by NASA's Edison Demonstration of SmallSat Networks, TanDEM-X, PROBA-3, and Prisma from ESA, and DARPA's System F6.



To understand the application isolation challenges concretely, let us consider an example of a satellite cluster. This cluster needs a distributed application that coordinates the orbits across all the CubeSats ensuring they do not collide. This safety-critical cluster flight application (CFA) can also make the satellites scatter away from each other if it detects a threat. Running concurrently with the CFA, image processing applications (IPA) utilize the satellites' sensors and consume much of the CPU resources. IPAs from different vendors may have different security privileges and so may have controlled access to sensor data. Sensitive camera data must be compartmentalized and must not be shared between these IPAs unless explicitly permitted. These applications must also be isolated from each other to prevent performance interference impact or fault propagation between applications due to lifecycle changes. However, the isolation should not waste CPU resources when critical applications are dormant because, for example, a sensor is active only in certain segments of the satellite's orbit. Other applications should be able to opportunistically use the computing resources during these dormant phases. That is, the highly constrained resources of the cluster should not be idly wasted if there are applications waiting for such resources. Additionally, the applications should be allowed to communicate only on authorized channels so that they cannot be used for receiving and send unauthorized information. This is one of a key requirement if the platform must be used in any critical mission.

One technique for implementing strict application isolation is temporal and spatial partitioning of processes (see [8]). Spatial separation provides a hardware-supported, physically separated memory address space for each process. Temporal partitioning provides a periodically repeating fixed interval of CPU time that is exclusively assigned to a group of cooperating tasks. Note that strictly partitioned systems are typically configured with a static schedule; any change in the schedule requires the system to be rebooted [8]. However, such strict isolation is only available in commercial ARINC-653 implementations and is typically too expensive for shared edge computing deployments like CubeSats. Thus, we have developed an operating system called COSMOS based on Linux. COSMOS is an extension of the DREMS-OS described in [9] called DREMS [7]. Extending the description in [9], in this paper, we describe the full architecture of the operating system including the secure communication layer, which ensures that the applications remain isolated, and the health management architecture, which ensures that the operating system can handle errors and recover upon detecting those errors.

The outline of this paper is as follows. We start by providing a description of Distributed Real-time Managed Systems in Section DISTRIBUTED REAL-TIME MANAGED SYSTEMS. Then, we discuss the operating system layer in Section COSMOS: THE OPERATING SYSTEM LAYER. This operating system is an extension of the system, which was described briefly in our prior work [9]. Specifically, we discuss the scheduling and CPU resource cap, a feature to provide work conserving behavior in Section SYSTEM SCHEDULER CONCEPTS. Thereafter, we describe the security concepts implemented in the operating system in Section SECURITY ARCHITECTURE. Health management and Fault Management architecture is discussed in Section HEALTH MANAGEMENT AND FAULT MANAGEMENT (HMFM) SUBSYSTEM. We finally conclude with three examples.

DISTRIBUTED REAL-TIME MANAGED SYSTEMS

Distributed real-time managed systems (DREMS) consist of multiple coordinated computing resources with several embedded system applications running on the resources. Our implementation of this architecture described in [10] consists of two major subsystems: 1) a design-time toolsuite for modeling, analysis, synthesis, implementation, debugging, testing, and maintenance of application software built from reusable components, and 2) a run-time software platform for deploying, managing, and operating application software on a network of computing nodes. The platform is tailored toward a managed network of computers and distributed software applications running on that network: a cluster of networked nodes.

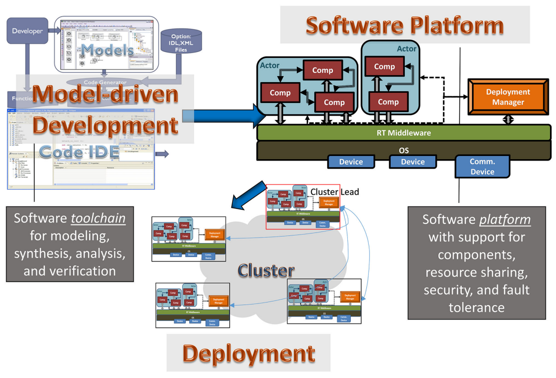


Figure 1.
The distributed real-time managed systems architecture.

Software applications running on the DREMS platform (see figure 1) are distributed: an application consists of one or more *actors* that run in parallel, typically on different nodes of a network. Actors specialize the concept of processes: they have identity with state, they can be migrated from node to node, and they are managed remotely. Actors are created, deployed, configured, and managed by a special service of the run-time platform: the deployment manager—a privileged, distributed, and fault tolerant actor, present on each node of the system, that performs all management functions on application actors. An actor can also be assigned a limited set of resources of the node it runs on: memory and file space, a share of CPU time, and a share of the network bandwidth. The operating system, which is the focus of this paper, implements all the critical low-level services to support resource sharing (including spatial and temporal partitioning), actor management, secure (labeled and managed) information flows, and fault tolerance.

COSMOS: THE OPERATING SYSTEM LAYER

COSMOS is a real-time operating system that allows system integrators to configure and execute processes of applications on a single computing node within the DREMS architecture. The core capabilities provided by the operating system are strong resource isolation and secure communication capabilities. The basis of that approach is the partitioning services baked into the operating system. These partitions enable cooperating processes called *actors* to work collaboratively, while being limited by the isolation policies by design. Processes within a partition execute concurrently with each other and are scheduled according to their associated priority. The priority of a process can be modified by any process in the partition (including itself). Processes can also suspend and resume other processes and enable or disable scheduling preemption. Within a partition, processes communicate with each other using four

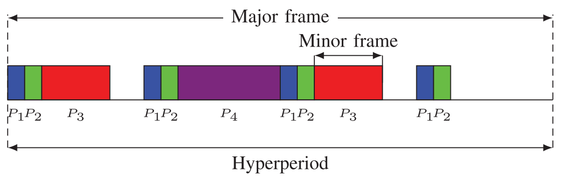


Figure 2.
Major frame. The four partitions (period, duration) in this frame are P_1 (2 s, 0.25 s), P_2 (2 s, 0.25 s), P_3 (4 s, 1 s), and P_4 (8 s, 1.5 s).

mechanisms: Blackboards (shared memory), Buffers (queues), Events, and Semaphores.

Temporal isolation in COSMOS is provided by the partition scheduler, which determines the time slots in which each partition can run. A continually repeating slice of time called a *hyperperiod* is divided into one or more nonoverlapping *minor frames*. A sequence of minor frames is called a major frame. Each minor frame consists of an offset from the start of the hyperperiod and a duration, along with one assigned partition that runs during this period. These minor frames can be of different durations and there can (and should) be gaps of time between the partitions to allow time for background operations of the kernel to run. This schedule is also defined in the configuration file and is fixed at run time. See Figure 2 for reference.

Each partition is managed by a separate Linux kernel thread and is assigned a unique name and an integer identifier (ID). An ID of 0 is reserved for the system partition, which is the initial thread that configures and launches the user-level partitions. The code written for each partition executes in user mode only—no privileged capabilities are permitted. In addition to spatial and temporal partitioning, each partition also has its own view of the file system, which means it has a unique file system directory from which it runs in a `chroot` jail, as well as its own storage allocation quota, file descriptors, access rights, storage device identification, and volume name aliasing.

Runqueues and System Partition: The *system* partition (ID 0) is created and scheduled along with other partitions to provide the system capability to schedule management tasks, for example, system-level health management. To support the different levels of criticality (both regular partitions and the System partition), we extend the *runqueue* data structure of the Linux kernel [11]. A runqueue maintains a list of tasks eligible for scheduling. In a multicore system, this structure is replicated per CPU. In a fully preemptive mode, the scheduling decision is made by evaluating which task should be executed next on a CPU when an interrupt handler exits, when a system call returns, or when the scheduler function is explicitly invoked to preempt the current process. We created one runqueue per temporal partition per CPU. Currently, the system can support 64 *Application partitions*. One extra runqueue is created for the tasks in the *System partition*. The OS also supports best effort tasks per partition. These tasks are managed through the

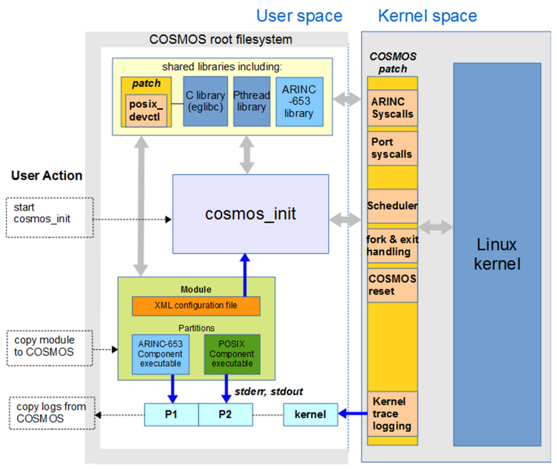


Figure 3.
Different components of a COSMOS module. The linux kernel is isolated from the user space by special ARINC-653 and POSIX system calls defined in the Future Airborne Capability Environment Standard [12]. Each module consists of a configuration file that defines the module schedule and the partitions. Upon reset, the partition logs can be copied over for introspection. The user-space application programs interact with the operating system using the C and ARINC 653 [8] libraries.

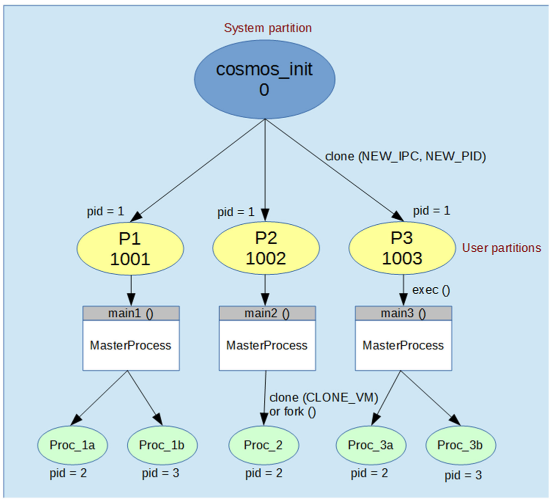


Figure 4.
Partition and process hierarchy in COSMOS operating system. The partitions and process show are just an example. The cosmos_init is the master process which spawns all the partition and starts a master process. The master process within each partition launches individual partition. Once all partitions are created the module state is set to *NORMAL*.

Linux Completely Fair Scheduler (default) runqueue (one per partition) and are considered for execution during a partition's time slice if no real-time task is eligible to be run.

SYSTEM COMPONENTS AND INITIALIZATION

Figure 3 shows the concise view of the system components and how it builds on top of the standard Linux kernel. cosmos_init is a system script responsible for setting up the partitions. It runs within the system partition. Figure 4 shows cosmos_init running as a system partition and creating the other user partitions based on the specifications in the configuration file, which it sets up as per partition data structures in the kernel. Thereafter, each partition is created using the kernel clone command to create a new process. It uses the CLONE_NEWIPC and CLONE_NEWPID options to indicate that this process is to use a separate interprocess communication namespace and PID namespace from the parent. CLONE_NEWIPC provides isolation for each partition when using System V IPC objects and POSIX message queues by only allowing other members of the namespace to access those objects. CLONE_NEWPID isolates the PID assignments in each partition such that the numbering will start with 1. The master process can use further fork calls to start up new process within the partition.

Once all partitions are set, cosmos_init completes its setup and launches the module by setting the module's state to *NORMAL*. It now performs a loop of waitpid() operations using the pid of the master process for each partition that was started. This call is invoked with WNOHANG to return immediately with the process status to determine if any of the master processes have terminated. When a master process terminates, it terminates all the other processes in the partition to shut the partition down and reclaim all of its resources. The death of a master process indicates that its partition has terminated. The exit code and signal are returned by the process to this waitpid() call, which can then identify the reason for the termination.

Table 1 summarizes the key concepts in the operating system and describe their Linux analog.

SYSTEM SCHEDULER CONCEPTS

In this section, we describe the key concepts of the COSMOS operating system. We specifically focus on the CPU resource cap, the partition scheduling loop, and the ability to configure the module's partition schedule without resetting the module.

CPU RESOURCE CAP

Once the module is configured by the cosmos_init process, it is set to the *NORMAL* state which enables it to start scheduling the tasks of each partition. However, to ensure that there is still some time for the system partition (hosting system level critical tasks) to run, we configure an upper limit called *CPU Cap* for all regular partitions. Specifically, the cap enables us to provide scheduling fairness within a partition (measured across a hyperperiod). Between criticality levels (application and system

Table 1.

Summary of Key Terms Used in the COSMOS Operating System

Process	A process in the COSMOS operating system is similar to a thread in Linux.
Actor	An actor is a collection of processes implemented by a vendor for some application.
Partition	A partition is a temporal slice where different actors of applications at similar criticality level can be collocated
Application	An application is a group of partition definitions and actor definitions provided by a vendor for a specific goal.
Module	A module is a computing node. It is configured using a configuration file.

partitions), the CPU cap provides the ability to prevent higher criticality tasks from starving lower criticality tasks of the CPU. At the *Application* level, the CPU cap is used to bound the CPU consumption of higher priority tasks to allow the execution of lower priority tasks inside the same partition. If the CPU cap enforcement is enabled, then it is possible to set a maximum CPU time that a task can use, measured over a configurable number of major frame cycles.

The CPU cap resource for a task is converted into a ceiling of execution time, which is measured over N major frames. The number of major frames over which the CPU cap ceiling is calculated is configurable at compilation time. When CPU cap is enabled, the scheduler maintains a counter for all tasks. The scheduler also maintains the current execution time of each task since the start of the CPU cap window. Note that at the beginning of each CPU cap window, the execution time of the task is reset to zero whereas the counter is reset after the number of major frame cycles over which the cap was specified elapses.

The counter is used when making a scheduling decision, which requires consideration of whether a task is ready and whether the task has surpassed its CPU cap quota. At every invocation of the scheduler, the execution time for the task is updated and compared against the execution time ceiling of the currently running task. If the task has surpassed its CPU cap quota within the CPU cap window, its $disabled$ flag is set to $true$.

The CPU cap is enforced in a work conserving manner, i.e., if a task has reached its CPU cap but there are no other available tasks, the scheduler will continue scheduling the task past its ceiling. In case of *Critical* tasks, when the CPU cap is reached, the task is not marked ready for execution unless 1) there is no other ready task in the system; or 2) the CPU cap accounting is reset. This behavior ensures that the kernel tasks, such as those belonging to network communication, do not overload the system, for example, in a denial-of-service attack. For the tasks on the *Application* level, the CPU cap is specified as a percentage of the total duration of the partition, the number of major frames, and the number of CPU cores available all multiplied together. When an *Application* task reaches the CPU cap, it is not eligible to be scheduled again unless the following is true: either there are no *Critical* tasks to schedule and there are no other ready tasks in the partition or the CPU cap accounting has been reset.

MAIN SCHEDULING LOOP

A periodic “tick” running at 250 Hz (The kernel tick value is also called “jiffy” and can be set to a different value when the kernel image is compiled.) is used to ensure that a scheduling decision is triggered at least every 4 ms. This tick runs with the base clock of *CPU0* and executes a procedure called $GlobalTick$ in the interrupt context only on *CPU0*. This procedure enforces the partition scheduling and updates the current minor frame and hyperperiod start time (HP_start). The partition schedule is determined by a circular linked list of minor frames which comprise the major frame. Each entry in this list contains that partition’s duration, so the scheduler can easily calculate when to switch to the next minor frame.

After the global tick handles the partition switching, the function to get the next runnable task is invoked. This function combines the *mixed criticality* scheduling with the *temporal partition* scheduling. For mixed criticality scheduling, the *Critical* system tasks should preempt the *Application* tasks, which themselves should preempt the *Best Effort* tasks. This policy is implemented by the *Pick_Next_Task* subroutine, which is called first for the system partition. Only if there are no runnable *Critical* system tasks and the scheduler state is not inactive, i.e., the application partitions are allowed to run (The OS provides support for pausing all application partitions and ensuring that only the system partition is executed.) will *Pick_Next_Task* be called for the *Application* tasks. Thus, the scheduler does not schedule any *Application* tasks during a major frame reconfiguration. Similarly, *Pick_Next_Task* will only be called for the *Best Effort* tasks if there are both no runnable *Critical* tasks and no runnable *Application* tasks.

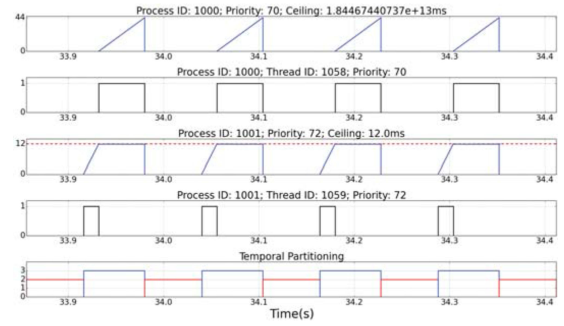


Figure 5.
Example: Single threaded processes 1000 and 1001 share a partition with a duration of 60 ms. Process 1000 has 100% CPU cap and priority 70; process 1001 has 20% CPU cap, and higher priority 72. Since process 1001 has a CPU cap less than 100%, a ceiling is calculated for this process as follows: 20% of 60 ms = 12 ms. The figure shows the example execution of this schedule. The average jitter was 2.136 ms with a maximum jitter of 4.0001 ms.

The *Pick_Next_Task* function returns either the highest priority task from the current temporal partition (or the system partition, as application) or an empty list if there are no runnable tasks. If CPU cap is disabled, the *Pick_-Next_Task* algorithm returns the first task from the specified runqueue. For the best effort class, the default algorithm for the Completely Fair Scheduler policy in the Linux kernel [13] is used.

If the CPU cap is enabled, the *Pick_Next_Task* algorithm iterates through the task list at the highest priority index of the runqueue because unlike the Linux scheduler, the tasks may have had their disabled bit set by the scheduler if it had enforced their CPU cap. If the algorithm finds a disabled task in the task list, it checks to see when it was disabled; if the task was disabled in the previous CPU cap window, it re-enables the task and sets it as the $next\_task$. If, however, the task was disabled in the current CPU cap window, the algorithm continues iterating through the task list until it finds a task which is enabled. If the algorithm finds no enabled task, it returns the first task from the list if the current runqueue belongs to an application partition.

This iteration through the task list when CPU cap enforcement is enabled increases the complexity of the scheduling algorithm to $O(n)$, where n is the number of tasks in that temporal partition, compared to the Linux scheduler's complexity of $O(1)$. Note that this complexity is incurred when CPU cap enforcement is enabled and there is at least one actor that has partial CPU cap (less than 100%). In the worst case, if all actors are given a partial CPU cap, the scheduler performance may degrade necessitating more efficient data structures.

To complete the enforcement of the CPU cap, the scheduler updates the statistics tracked about the task and then updates the disabled bit of the task accordingly. Figure 5 shows the above-mentioned scheduler decisions when CPU cap is placed on processes that share a temporal partition. To facilitate the analysis, the scheduler uses a logging framework that updates a log every time a context switch happens. Figure 5 clearly shows the lower priority actor executing after the higher priority actor has reached its CPU cap.

DYNAMIC MAJOR FRAME CONFIGURATION

One of the innovations in this architecture is that the initial configuration process (described earlier) can be repeated at any time without rebooting the node: the kernel receives the major frame structure that contains a list of minor frames and it also contains the length of the hyperperiod, partition periodicity, and duration. Note that major frame reconfiguration can only be performed by an actor with suitable capabilities (controlled using the POSIX Capability model).

Before the frames are set up, the process configuring the frame has to ensure that the following three constraints are met: C0) The hyperperiod must be the least common multiple of partition periods; C1) The offset between the major frame start and the first minor frame of a partition must be less than or equal to the partition period: $(\forall p \in \mathbb{P})(O_1^p \leq \phi(p))$; C2) Time between any two executions should be equal to the partition period: $(\forall p \in \mathbb{P})(k \in [1, N(p) - 1])(O_{k+1}^p = O_k^p + \phi(p))$, where $\mathbb{P}$ is the set of all partitions, $N(p)$ is the number of partitions, $\phi(p)$ is the period of partition p, and $\Delta(p)$ is the duration of the partition p. O_i^p is the offset of ith minor frame for partition p from the start of the major frame and H is the hyperperiod.

The kernel checks two additional constraints: 1) All minor frames finish before the end of the hyperperiod: $(\forall i)(O_i.\text{start} + O_i.\text{duration} \leq H)$ and 2) minor frames cannot overlap, i.e., given a sorted minor frame list (based on their offsets): $(\forall i < N(O))(O_i.\text{start} + O_i.\text{duration} \leq O_{i+1})$, where $N(O)$ is the number of minor frames. Note that the minor frames need not be contiguous, as the update procedure fills in any gaps automatically.

If the constraints are satisfied, then the task is moved to the first core, *CPU0* if it is not already on *CPU0*. This is done because the global tick (explained in the next section) used for implementing the major frame schedule is also executed on *CPU0*. By moving the task to *CPU0* and disabling interrupts, the scheduler ensures that the current frame is not changed while the major frame is being updated. At this point, the task also obtains a spin lock to ensure that no other task can update the major frame at the same time. In this procedure, the scheduler state is also set to APP_INACTIVE (see Table 2) to stop the scheduling of all application tasks across other cores. The main scheduling loop reads the scheduler state before scheduling application tasks. A scenario showing dynamic reconfiguration can be seen in Figure 6.

Table 2.

States of the DREMS Scheduler	
APP_INACTIVE	Tasks in temporal partitions are not run
NORMAL	Inverse of APP_INACTIVE

It is also possible to set the global tick (that counts the hyperperiods) to be started with an offset. This delay can be used to synchronize the start of the hyperperiods across nodes of the cluster. This is necessary to ensure that all nodes schedule related temporal partitions at the same time. This ensures that for an application that is distributed across multiple nodes, its *Application* level tasks run at approximately the same time on all the nodes which enables low latency communication between dependent tasks across the node level.

SECURITY ARCHITECTURE

The basis of the security design in the system is the combination of discretionary access control over file resources, (The discretionary access control is the same as in Linux and implemented using user-ids associated with applications. We do not discuss it further as it is a direct usage of existing facilities.) strong isolation between computational resources used by different applications, mandatory access control and strict separation between communication mechanisms, essentially controlling the flow of information via multilevel security and a strong capability model that describes which operating services can be used by which application. We discuss the various security concepts below.

MANDATORY ACCESS CONTROL AND THE CAPABILITY MODEL

Our security architecture uses (multidomain) labels, an extension of the multilevel security (MLS) policy ([6], [14], [15]). MLS defines a policy based on partially ordered security labels that assign classification and need-to-know categories to all information that may pass across process boundaries. This policy requires that information only be allowed to flow from a producer to a consumer if and only if the label of the consumer is greater than or equal to that of the producer. Labels are organized in a lattice, where levels indicate classification. These labels are often assigned by a trusted party and the labels are associated with each actor using the process control data structure maintained in the kernel. Only processes with appropriate privileged are allowed to modify the data structure and the data structure can be modified only at the time of the creation of the actor. Any subsequent modification requires restarting the actor.

In addition, the operating system divides the system calls into different categories and partitions the categories into privileged and regular calls. The cartesian product of this set results in 48 different levels in the operating system. These levels are encoded in a 64-bit integer and assigned to each actor at the start. Only the cosmos_init process described earlier has the create actor privilege and only that process can assign the capabilities to other actors. An actor cannot change its privilege level. When a system call is made the OS checks the identity of the actor against the capability mask and decides whether the call should go forward or not. This static rigid model of assigning capability ensures that even when compromised, an actor cannot execute the calls it is not allowed to. By default, no application actor is given access to any privileged call. Privileged system calls may result in the modification of nonprocess related kernel data structures, for example, it can update the major frame (see Section "Dynamic Major Frame Configuration").

SPATIAL, STORAGE AND TEMPORAL SEPARATION

The spatial separation of volatile memory is important from a security perspective because it prevents an actor from accessing other actors' or the operating system's volatile data. Since actors are often implemented in C/C++, which is not type-safe, this spatial separation would be difficult to guarantee by analysis of the actors' code. The spatial separation is maintained by strict physical memory boundaries if no swap memory is present. If swap memory is present, then traditional virtual memory separation is used in addition with a strict virtual memory quota that can never be exceeded.

To ensure separation in persistent storage, each actor is assigned a separate file system for persistent data

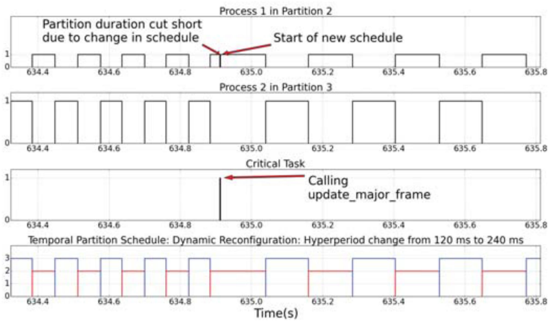


Figure 6.
Dynamic major frame reconfiguration example: Two single-threaded processes run in separate partitions with a duration of 60 ms each. The schedule is dynamically reconfigured so that each partition duration is doubled. A *Critical* task is responsible for calling the update_major_frame system call. Duration of the active partition is cut short at the point when update_major_frame function is called.

storage. Since each actor's private file system is invisible to other actors (except the platform processes with privileged storage capability), files need not be labeled for mandatory access control or decorated by access control lists for discretionary access control. Communication among actors only takes place via messages that go through the operating system using the secure transport (described later), not via shared files or shared memory.

The temporal separation is provided by the partitions. The temporal separation is important from a security perspective because it prevents covert timing channels across different temporal partitions. More than one actor can be assigned to the same temporal partition, and our design does not prescribe a specific approach to assign actors to temporal partitions. Typically, actors of the same application may be added to one temporal partition. Actors running in the privileged system partition (implementing critical services) can run on demand (e.g., when a system call is performed) during other actors' temporal partitions (the preempt all other actors); they are not assigned to any temporal partition but have their time "charged" to the requesting actors.

COMMUNICATION VIA SECURE TRANSPORT

To support communication and coordination between applications of different criticality, priority, and security levels, we developed the secure transport (ST) facility. ST is a managed communications infrastructure that provides for datagram oriented exchange of messages between application tasks. ST restricts the transmission of datagrams according to both a communication topology and a mandatory access control policy described above.

Actors communicate through endpoints connected by flows. A flow connects one source endpoint to one or more destination endpoints. Destination endpoints can be local (i.e., on the same node as the source endpoint) or remote (i.e., on different nodes). Like traditional sockets, user-space programs pass an endpoint identifier to the `send` and `receive` system calls. Unlike traditional sockets, however, unprivileged tasks may not arbitrarily construct endpoints that allow for interprocess communication with other tasks; such endpoints must be explicitly configured by a privileged task acting as a trusted system configuration authority. When an endpoint is created, the operating system ensures that endpoints are preconfigured with labels and that the label of an endpoint is a strict subset of the labels of the actor. Actors choose a label to be associated with a message when sending it. The flow and the label restrict the possible destinations. In the operating system, three kinds of endpoints are supported.

- *Local Message Endpoints (LME)*: Local message endpoints are the basic method of IPC, and may be used to send messages to other tasks hosted by the same operating system instance. These endpoints must be configured by trusted system configuration infrastructure and are subject to restrictions placed by flows and security rules.
- *Kernel Message Endpoints (KME)*: Kernel message endpoints are like netlink (Netlink sockets are a way to transfer information between the kernel and user-space processes in Linux.) sockets and are used to communicate with the kernel. The only difference is that the KMEs have a strict memory limit and the memory limit is charged against the memory quota of the actor using the endpoint.
- *Remote Message Endpoints (RME)*: Similar to LMEs, RMEs are a mechanism for IPC between tasks, but may be used to communicate with tasks hosted by different operating system instances throughout a network.

Flows: In ST, communication is allowed between two LME or RME endpoints if and only if there exist mutually compatible *flows* on each endpoint. A flow, assigned to an endpoint, is a connectionless association with an endpoint owned by a designated process (in *DREMS*, process identifiers are statically assigned and globally unique). This association determines if the local endpoint can send or receive messages with the remote endpoint. In all cases, flow assignment between two endpoints must be mutual in order for communication to succeed, i.e., a sender must have an outbound flow to the recipient, and the recipient must have an inbound flow from the sender. Further, the flow should conform to the security policies.

Sending Messages: System calls to send messages require the calling actor to specify, among other things, a label and an endpoint. The OS checks the label of the message against the labels of that endpoint, which, as described above, are securely stored inside the kernel data structures and are known to be a subset of the labels of the actor as described earlier. If the message label is not among the labels of the endpoint, the message is not sent (in the case of a write endpoint) or delivered (in the case of a read endpoint), but instead it is discarded. These checks are performed by the OS and cannot be bypassed.

When an actor attempts to send a message to a write endpoint, the OS checks the label of the message not only against the endpoint's labels, but also against the labels of the destination endpoints connected by the flow to the source write endpoint. Even though the label is checked at the actual destinations, this preemptive check avoids sending a message (on the network) whose label would fail the check at the destination. If actors are properly configured across the cluster, each endpoint will be assigned the same labels locally (where the endpoint resides) and remotely (in other nodes that include flows with that endpoint as destination). If for any reason there is an inconsistency in these assignments, a legitimate message may get blocked, but an illegitimate message will never be delivered.

Table 3.

Health Management Tables Specify the Mapping of Errors at Different Levels to the Actions that Must be Performed	
Module Level Table	Errors listed at this level will cause one of the MODULE level actions to occur. If this table is traversed and the error is not found, the default action is to shut down the module. Only one Module table can be defined for the module. If it is omitted, all errors that occur outside of partition space will cause a module shutdown.
Multipartition Table	If a partition error requires module level mitigation then the table at this level is used. More than one Multi_Partition_HM_Table is allowed so that each partition can have unique error handling for it. Each table is given a name and each partition specifies the Multi_Partition_HM_table that is to be used for it in the configuration file (Figure 3). If it is not specified, all partition errors default to the Partition_HM_table specified for it.
Partition level Table	The errors in this table can be either at the partition level or the process level. All listed errors in this table specify a partition level action. Process level errors define a corresponding ARINC-653 error code value, which is sent to an error handler process, if available within the partition. If an error handler process is unavailable then a specified default action is executed.

Some partition level errors (if specified) are handled by a high priority error handling process within the partition. This ensures that no other partitions are affected if a process error occurs.

HEALTH MANAGEMENT AND FAULT MANAGEMENT (HMFM) SUBSYSTEM

The health management framework in the COSMOS operating system is similar to the ARINC-653 specification [8]. It can handle errors at three levels: Module, Partition, or a Process with a set of defined mitigation actions. The errors at the module level occur outside the scope of any partition. For example, during startup of the system, the system supports following mitigation actions for the module-level errors: a) Ignore the error, b) shutdown the module, c) restart the module. The partition-level errors are of those that cannot be resolved within the scope of a single process and require the following types of corrective actions: a) Ignore the error, b) stop the partition, c) restart the partition. The process level errors are handled in the user space by invoking preregistered error handling functions. The error to mitigation action configuration is stored in three tables described in Table 3.

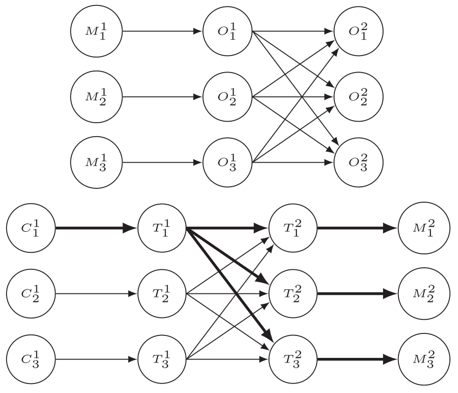
The health monitoring tables are setup during the system initialization by the cosmos_init process. Once the master process of the partition has finished initialization (see Section “System Components and Initialization”) it goes into a blocking state waiting on a local message endpoint (see Section ”Communication via Secure Transport”) to receive any error message raised by a process within the partition. The master process of the partition then checks if the error is partition specific (i.e., it can be handled in user space) or if it needs to be handled at the partition or the module level. Errors at that level are then handled by the error handling process (one per partition) in the kernel. The partition master process activates the error handler process by sending it a message on another local message endpoint.

ERROR HANDLING PROCESS

The error handling process runs at the highest priority in the partition and first determines if the local partition level error is fatal, requiring that the current partition be reset. If so, it does this immediately. An action to reset the partition is handled by the error handler process, while remaining within the temporal bounds set by the partition schedule.

The error handling process follows the following protocol to search the error to action mapping configured in the tables (see Table 3). If the error has occurred outside the partition space, then the table is searched using the error id and a table name, which specifies the module-level table entries. If the error is not found in the table, then the default action is performed. If the error occurred during the execution of a partition process, then it first finds the entry of the error in the associated multipartition table. If a multipartition table entry is defined, the table search key will consist of this table name and the error id, which will return either a module level action (the default action if the error was not found), or indicate it is a partition level error and provides a different error id.

If no multipartition table is defined for the partition or the table states that the error is a partition level error then the search continues using a search key consisting of the partition table name and the error id. If the error is not found or no partition table entry is defined for the partition, the default action (specified in the module configuration file) will again be taken.



Processes	Actor	Activity
M^1	$ModuleProxy$	Inform O^1 of new state
M^2		Activate engine
O^1	$OrbitalMaintenance$	Publish new state
O^2		Subscribe to new state
T^1	$TrajectoryPlanning$	Publish new command
T^2		Subscribe to new command
C^1	$CommandProxy$	Inform T^1 of command

Figure 7.
ModuleProxy actor controls thruster activation in Orbiter and state vector retrieval from Orbiter. *OrbitalMantenance* actor tracks the cluster satellites' state vectors and disseminate them. *TrajectoryPlanning* actor controls the response to commands and satellite thruster activation. *CommandProxy* actor informs the satellite of a command from the ground network. Each actor has multiple processes. The subscript represents the node ID on which the process is deployed. The superscript denotes the different jobs done by an actor. Thus, M_2^2 is the process that activates the engine of node 2 and M_1^2 is the process that informs about the current satellite state to the orbital maintenance actor. Of particular interest to safety of the constellation is the control flow between *CommandProxy* actor and the *ModuleProxy* actor process on each node responsible for thruster activation. This interaction pathway is in bold.

EXAMPLES AND EVALUATION

To evaluate the different aspects of the operating system, we now describe three examples. The first example is an application focusing on the scheduling and secure transport overhead. The second and third examples focus on the performance of the health management and fault management subsystem.

CLUSTER FLIGHT APPLICATION

We created a multicomputing node experiment on a cluster of fan less computing nodes with a 1.6-GHz Atom N270 processor and 1 GB of RAM each. We chose these nodes to emulate the embedded processors typically used in hash/rugged military and space applications. On these nodes, a cluster of three satellites was emulated. These satellites ran two distributed applications: a CFA, and an image processing application.

The CFA (see Figure 7) consists of four actors: *OrbitalMaintenance*, *TrajectoryPlanning*, *CommandProxy*, and *ModuleProxy*. *ModuleProxy* connects to the Orbiter space flight simulator (http://orbit.medphys.ucl.ac.uk/) that simulates the satellite hardware and orbital mechanics for the three satellites flying in low Earth orbit. *CommandProxy* receives commands from the ground network. *OrbitalMaintenance* keeps track of every satellite's position and updates the cluster with its current position. This is done by a publish subscribe interaction between all Orbital Maintenance actors. Note that we have not discussed the specific middleware level interaction patterns implemented in our design. However, interested readers can refer to [16] and [17].

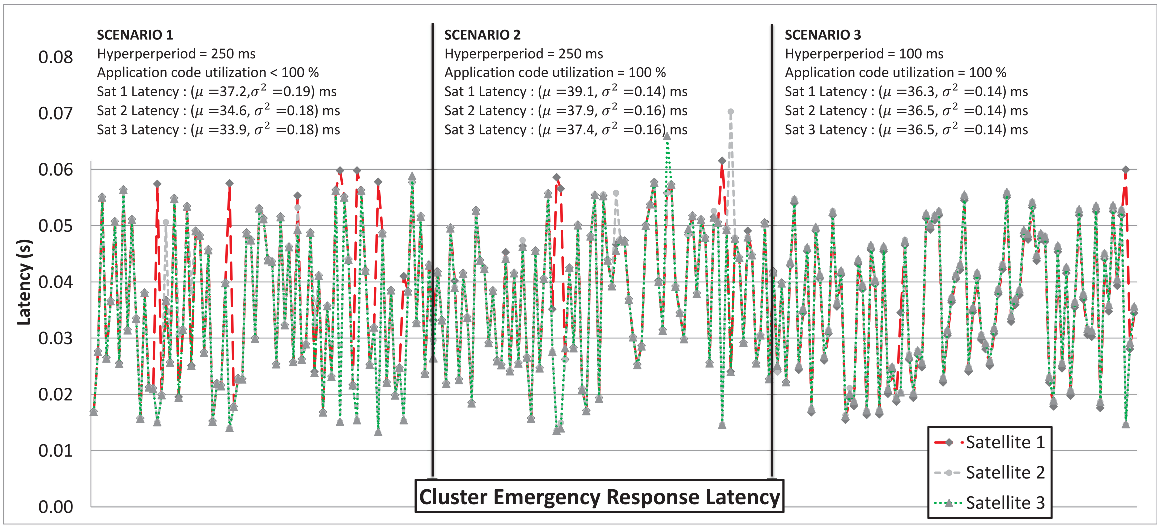
The image processing application workload was simulated using four image processing actors (IPA) per node that can be configured dynamically to load the machines. We assigned two image processing actors to the first partition and two image processing actors to the second partition. The third shorter partition was reserved for the *OrbitalMaintenance* actor. It is a periodic process that publishes and subscribes to the satellite state every second and is not critical in an emergency. The other CFA actors are assigned to the system partition as they are deemed critical and need to run as soon as possible. Further, since only the cluster leader (node *1*) receives the scatter command, the *CommandProxy* actors are not active one the second and third nodes.

Figures 8(a) and (b) show the results from three different scenarios: 1) hyperperiod of 250 ms, with IPA consuming less than 50 percent CPU, 2) hyperperiod of 250 ms, with IPA consuming 100 percent CPU, and 3) hyperperiod of 100 ms, with IPA consuming 100 percent CPU. (These latencies were calculated from time-stamped messages collected in the application logs. All the nodes' clocks were synchronized to within 10 μs.) The first thing to note is that Figure 8(b) demonstrates the proper preemption of the image processing actor process by the critical CFA actor process for scenario 2 (hyperperiod of 250 ms, with IPA consuming 100 percent CPU).

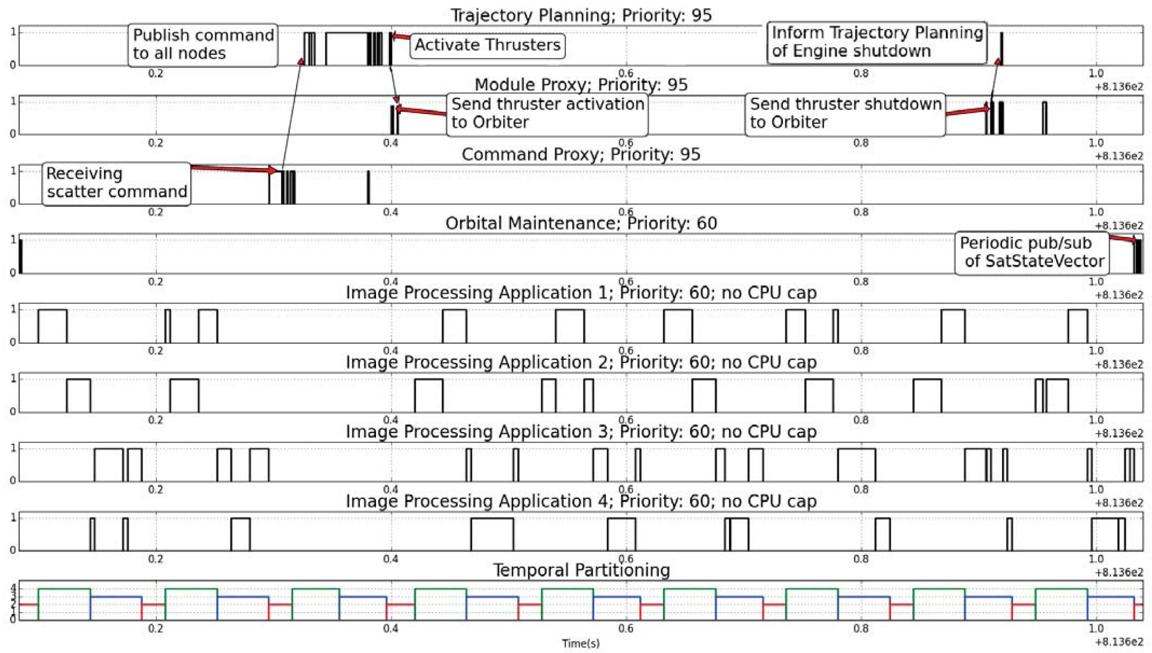
Figure 8(a) shows that the emergency response latency over the three nodes was quite low with very little variance, and did not correlate with either the image application's CPU utilization or the application's partition schedule. As such, the satellite cluster running the *DREMS* infrastructure is able to quickly respond to emergency situations despite high application CPU loads and without altering the partition scheduling.

HEALTH MONITORING AND FAULT MANAGEMENT EVALUATION

We used a one partition setup with a simple send–receive application to evaluate the delay times associated with responding to an health management event and the time it



(a) This is the time between reception of the *scatter* command by satellite 1 and the activation of the thrusters on each satellite corresponding to interactions $C_1^1 \rightarrow M_N^2$ of Figure 7. The three regions of the plot indicate the three scenarios: (1) image processing application has limited use of its partitions and has a hyperperiod of 250 ms, (2) image processing application has full use of its partitions and has a hyperperiod of 250 ms, and (3) image processing application has full use of its partitions and has a hyperperiod of 100 ms. The averages and variances for the satellites' latencies are shown for each of the three scenarios.



(b) The engine activation following reception of a *scatter* command is annotated for the relevant actors for scenario 2. The *scatter* command causes the *TrajectoryPlanning* to request *ModuleProxy* to activate the thrusters for 500 ms. The image processing does not run while the mission-critical processes are executing - without halting the partition scheduling. The context switching during the execution of the critical processes is the execution of the secure transport kernel thread. Only the application processes are shown in the figure; the kernel threads and other background processes have been left out for clarity.

Figure 8.
DREMS mixed criticality demo.

takes to process the correction for it. The average time (for 50 errors) it took to detect the error condition was 39.19 ms and the time for the error handler process to complete the action of shutting down and restarting the failed process was 101.84 ms. These can be seen graphically in Figure 9. This figure shows the time along the vertical axis in milliseconds, with the horizontal axis indicating the error occurrence. The blue graph at the bottom shows the time it takes from the occurrence of an error to when the master process wakes from receiving the error message. The red graph above it indicates the amount of time elapsed between the master process receiving the notification to

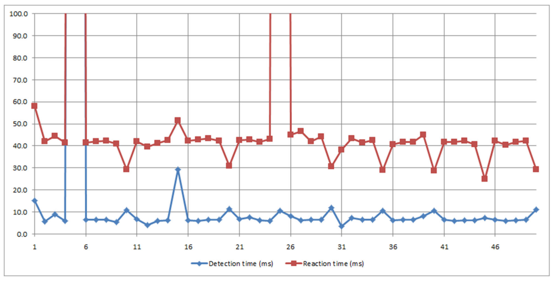


Figure 9.
Detection and reaction time over ten iterations for the one partition health monitoring example.

when the Error Handler process has completed the specified actions and has exited. Deadline-related anomalies were caught by the scheduler within 7.72 ms. It took on average 40.83 ms for restarting the process due to the deadline violation.

TWO PARTITION HMFM EXAMPLE

Sometimes there are additional delays in processing an error condition due to the temporal partitioning, as is demonstrated at error 5 and error 25 in Figure 9. A two partition example (see Figure 10) illustrates this point. In this example, there are two partitions: *P1* and *P2*. Each partition runs for a duration of 400 ms within a 1000 ms hyperperiod. Partition P2 gets scheduled to run by the kernel at point *A* in the diagram. The first process that runs is *ACCUM* which performs some numeric analysis on data it receives from another higher priority process. It performs initialization and then starts the second process, *RCVR* which begins immediately due to its higher priority.

The *RCVR* process then experiences a memory fault due to an out-of-bounds access that is shown at point *B*. The kernel catches the error. The signal in this case is *SIGSEGV*, which corresponds to the error id HMFM_MEM_VIOLATION. Because this is a fatal error code, it is handled immediately by the error handling process. It checks if a partition is currently running and looks up the name of the partition and the multipartition table that it is using. The error is defined as a process level error that is to be handled by the error handler defined for this partition. It then sends a message to the local endpoint of the master process. This causes the master process to run since it has the highest priority assigned to it and is shown as point *C*.

A few milliseconds later, at point *D*, the minor frame 2 reaches the end of its 400 ms duration. Even though the master process is running at the highest priority, the kernel scheduler halts the process to begin the next scheduled event. There is a 100 ms gap where system processes can now run followed by partition $P1$ and another 100 ms gap. After partition $P2$ 600 ms was suspended, it is again scheduled to run for another 400 ms at point E. The master process immediately runs and sends a $START$ signal to the error handler process. The error handler process begins running at F and continues until it completes at point G. During this time, it restarts the $RCVR$ process, which then continues to run and receives input on port 2 from partition $P1$. It then sends this data internally to the $ACCUM$ process and then blocks waiting for more input from the port, allowing $ACCUM$ to read and process the data sent by $RCVR$. This illustrates that even though the response time of the Health Monitor is in the millisecond range, it may take another iteration of the hyperframe to complete the task if the partition has limited duration.

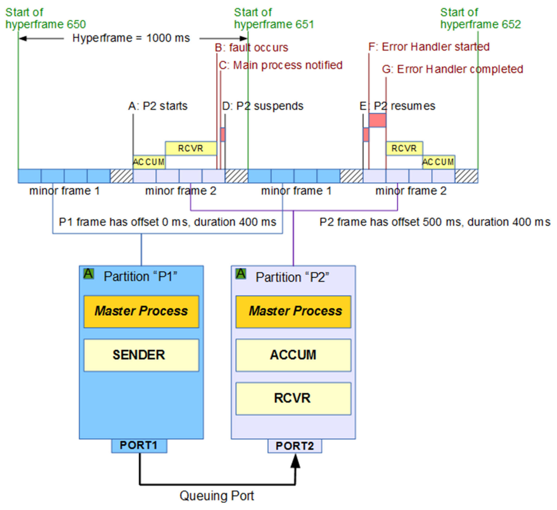


Figure 10.
Two partition health monitoring example.

RELATED RESEARCH

Our approach has been inspired by two domains: mixed criticality systems and partitioning operating systems. A mixed criticality computing system has two or more criticality levels on a single shared hardware platform, where the distinct levels are motivated by safety and/or security concerns. For example, an avionics system can have safety-critical, mission-critical, and noncritical tasks.

In [18], Vestal argued that the criticality levels directly impact the task parameters, especially the worst-case execution time (WCET). In their framework, each task has a maximum criticality level and a nonincreasing WCET for successively decreasing criticality levels. For criticality levels higher than the task maximum, the task is excluded from the analyzed set of tasks. Thus, increasing criticality levels result in a more conservative verification process. The authors extended the response-time analysis of fixed priority scheduling to mixed criticality task sets. These results were later improved by Baruah et al. [19] where an implementation was proposed for fixed priority single processor scheduling of mixed-criticality tasks with optimal priority assignment and response-time analysis.

Partitioning operating systems have been applied to avionics (e.g., LynxOS-178 [20]), automotive (e.g., Tresos, the operating system defined in AUTOSAR [21]), and cross-industry domains (DECOS OS [22]). A comparison of the mentioned partitioning operating systems can be found in [23]. They provide applications with shared access to critical system resources on an integrated computing platform. Applications may belong to different security domains and can have different safety-critical impact on the system. To avoid unwanted interference between the applications, reliable protection is guaranteed in both the spatial and the temporal domains, which is achieved by using partitions on the system level. Spatial partitioning ensures that an application cannot access another application's code or data in memory or on disk. Temporal partitioning guarantees an application access to the critical system (e.g., CPU) resources via dedicated time intervals regardless of other applications.

Our approach combines mixed-criticality and partitioning techniques to meet the requirements of secure distributed real-time and embedded (DRE) systems. *DREMS* supports multiple levels of criticality, with tasks being assigned to a single criticality level. For security and fault isolation reasons, applications are strictly separated by means of spatial and temporal partitioning, and applications are required to use a novel secure communication method for all communications, which is described in *DREMS* [6].

This work has many similarities with the resource-centric, real-time kernel [24] to support real-time requirements of distributed systems hosting multiple applications. Though achieved differently, both frameworks use deployment services for the automatic deployment of distributed applications, and for enforcing resource isolation among applications. However, to the best of our knowledge, Lakshmanan and Rajkumar [24] do not include support for process management, temporal isolation guarantees, partition management, and secure communications simultaneously.

CONCLUSION

The *DREMS* platform consists of a development environment and runtime platform that provides an infrastructure to address several challenges faced in deploying and managing a cluster of mobile embedded platforms. It provides the capability for domain specific modeling and a novel real-time operating system with support for a distributed component model and several management services for controlled and managed deployment of applications on distributed computing nodes.

This paper propounds the notion of managed DRE systems that are deployed in mobile computing environments. These systems must be managed due to the presence of mixed criticality task sets that operate at different temporal and spatial scales, and share the resources of the DRE system. To that end, we have described the design and implementation of the next version for the *DREMS*operating system, including the scheduler, the secure communication mechanism, and a health management mechanism.

We also analyzed the scheduler properties of a distributed application built entirely using this platform and hosted on an emulated cluster of satellites.

To extend this work further, we are working on the response–time analysis on the task level, and on design–time analysis and verification tools for the component-level scheduler, which operates within each component for scheduling the component's operations. Additionally, such complex networked systems with mission-critical tasks distributed among many nodes require guarantees about the network QoS for each task needing access to the network. However, the temporal partitioning of the application tasks significantly affects task access both to the CPU and to network resources. Finally, a more comprehensive fault diagnostics and response infrastructure is needed for robust cluster performance in adverse situations.

ACKNOWLEDGMENTS

This work was supported by DARPA under Contract NNA11AC08C. Portions of COSMOS operating system were also developed under funding from NavAir. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of DARPA and NavAir. The authors would like to thank A. Coglio from Kestrel who implemented the multilabel security concept and also the contributions of W. Otte who implemented the secure transport. Details of that implementation are described in [10].

REFERENCES

[1] F. Sun, A. Dubey, C. Kulkarni, N. Mahadevan, and A. G. Luna, "A data driven health monitoring approach to extending small sats mission," in *Proc. Annu. Conf. Prognostics Health Manage. Soc.*, 2018, https://www.phmpapers.org/index.php/phmconf/article/download/573/phmc_18_573.

[2] M. García-Valls, A. Dubey, and V. Botti, "Introducing the new paradigm of social dispersed computing: Applications, technologies and challenges," *J. Syst. Archit.*, vol. 91, pp. 83–102, 2018.

[3] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2016, pp. 1–13.

[4] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[5] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *J. Syst. Archit.*, vol. 60, no. 9, pp. 726–740, 2014.

[6] A. Dubey et al., “A software platform for fractionated spacecraft,” in *Proc. IEEE Aerosp. Conf.*, Mar. 2012, pp. 1–20.

[7] G. Karsai, D. Balasubramanian, A. Dubey, and W. R. Otte, “Distributed and managed: Research challenges and opportunities of the next generation cyber-physical systems,” in *Proc. IEEE 17th Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput.*, Jun. 2014, pp. 1–8.

[8] *Document No. 653: Avionics Application Software Standard Interface (Draft 15)*, ARINC Incorporated, Annapolis, MD, USA, Jan. 1997.

[9] A. Dubey, G. Karsai, A. Gokhale, W. Emfinger, and P. Kumar, “DREMS-OS: An operating system for managed distributed real-time embedded systems,” in *Proc. 6th Int. Conf. Space Mission Challenges Inf. Technol.*, vol. 00, Sep. 2018, pp. 114–119. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SMC-IT.2017.26

[10] T. Levendovszky et al., “Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems,” *IEEE Softw.*, vol. 31, no. 2, pp. 62–69, Mar./Apr. 2014.

[11] A. Garg, “Real-time linux kernel scheduler,” *Linux J.*, vol. 2009, no. 184, p. 2, 2009.

[12] OpenGroup. [Online]. Available: http://www.opengroup.org/face. Accessed. 2019.

[13] W. Mauerer, *Professional Linux Kernel Architecture*, ser. Wrox professional guides. Wiley, 2008. [Online]. Available: http://books.google.com/books?id=4eCr9dr0uaYC

[14] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE, McLean, VA, USA, Tech. Rep. 2547, Volume I, 1973.

[15] O. Sibert, “Multiple-domain labels,” presented at the F6 Security Kickoff, 2011.

[16] A. Dubey, G. Karsai, and N. Mahadevan, “A component model for hard real-time systems: CCM with ARINC-653,” *Softw., Pract. Experience*, vol. 41, no. 12, pp. 1517–1550, 2011.

[17] W. R. Otte et al., “F6COM: A component model for resource-constrained and dynamic space-based computing environment,” in *Proc. 16th IEEE Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, Paderborn, Germany, Jun. 2013, pp. 1–8.

[18] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *Proc. 28th IEEE Real-Time Syst. Symp.*, Tucson, AZ, USA, Dec. 2007, pp. 239–243.

[19] S. Baruah, A. Burns, and R. Davis, “Response-time analysis for mixed-criticality systems,” in *Proc. 32nd IEEE Real-Time Syst. Symp.*, Vienna, Austria, Nov. 2011, pp. 34–43.

[20] LynuxWorks, “RTOS for Software Certification: LynxOS-178.” [Online]. Available: http://www.lynuxworks.com/rtos/rtos-178.php. Accessed 2019.

[21] Autosar GbR, “AUTomotive Open System ARchitecture,” [Online]. Available: http://www.autosar.org/. Accessed 2019.

[22] R. Obermaisser, P. Peti, B. Huber, and C. E. Salloum, “DECOS: An integrated time-triggered architecture,” *J. Austrian Professional Inst. Elect. Inf. Eng.*, vol. 123, no. 3, pp. 83–95, Mar. 2006.

[23] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, “A comparison of partitioning operating systems for integrated systems,” in *Computer Safety, Reliability and Security*. New York, NY, USA: Springer, 2007, vol. 4680/2007, pp. 342–355.

[24] K. Lakshmanan and R. Rajkumar, “Distributed resource kernels: OS support for end-to-end resource isolation,” in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, St. Louis, MO, USA, Apr. 2008, pp. 195–204.