DREMS: OS Support for Managed Distributed Real-time Embedded Systems

William Emfinger, Pranav Kumar, William Otte, Csanad Szabo,
Sandor Nyako, Abhishek Dubey, Aniruddha Gokhale and Gabor Karsai
ISIS, Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA

***Abstract*—Distributed real-time and embedded (DRE) systems composed of mixed criticality task sets are increasingly being deployed in mobile and embedded cloud computing platforms. These DRE systems not only must operate over a range of temporal and spatial scales, but also require stringent assurances on the secure communications between the system's tasks without violating timing constraints of the individual tasks. To address these new challenges this paper describes a novel distributed operating system. The paper focuses on the scheduler design to support the mixed criticality task sets, and a novel secure networking infrastructure. Empirical results from experiments involving a case study of a cluster of satellites emulated in a laboratory testbed validates our claims.**

Keywords*—Mixed criticality tasks, security, operating systems, managed distributed systems.

I. Introduction

The emerging realm of mobile and embedded cloud computing, which leverages the progress made in computing and communication on mobile devices and sensors necessitates a platform for running distributed real-time and embedded (DRE) systems. For example, an adhoc cloud of smart phones can share sensing and computing resources with nearby devices to provide increased situational awareness in disaster relief efforts. Ensembles of mobile devices are being used as a computing resource in space missions as well: clusters of satellites provide a dynamic environment for deploying and managing distributed mission applications; see, *e.g.* NASA's Edison Demonstration of SmallSat Networks, TanDEM-X, PROBA-3, and Prisma from ESA, and DARPA's System F6.

As an example consider a cluster of satellites that execute distributed applications. One application is a safety-critical cluster flight application (CFA) that controls the satellite's flight and responds to emergency commands. Running concurrently with the CFA, image processing applications (IPA) utilize the satellites' sensors and consume much of the CPU resource. IPAs from different vendors may have different security privileges and so may have limited 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 impact or fault propagation between applications due to lifecycle changes. However, the isolation should not waste CPU resources when 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 use the CPU during these dormant phases. Management of these applications entails providing for secure information flows, application lifecycle management, fault isolation, resource partitioning, and partition reconfiguration.

One technique for implementing strict application isolation is temporal and spatial partitioning of processes (see [1]). Spatial separation provides a separate memory address space per process. Temporal partitioning provides a periodically repeating fixed interval of CPU time that is exclusively assigned to a group of cooperating tasks. Unfortunately, strictly partitioned systems are typically configured with a static schedule; any change in the schedule requires the system to be rebooted [1].

We have developed an architecture called **D**istributed **RE**altime **M**anaged **S**ystem (*DREMS*) [2] that addresses these requirements. *DREMS* addresses a class of DRE systems that require active *management* of the software platform and the applications running on that platform. Such managed DREs arise in application domains where a dominant entity controls the complete software configuration and all operational aspects of a large number of computing nodes, shared by many distributed applications. The architecture consists of (1) a design-time tool suite for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components, and (2) a run-time *software platform* for deploying, executing, and managing application software on a network of mobile nodes. The run-time software platform consists of an operating system kernel, system services and middleware libraries. In prior work, we have described the general architecture of *DREMS* [2], its design-time modeling capability [3], and its component model used to build applications [4].

This paper focuses on the design and implementation of key components of the operating system layer in *DREMS*. Specifically, it makes the following three contributions to the realm of operating systems for managed DRE systems that operate in mobile and embedded computing environments.

- It describes the design choices and algorithms used in the design of *DREMS* OS scheduler. The scheduler supports three criticality levels: critical, application and best effort. It supports temporal and spatial partitioning for application-level tasks. Tasks in a partition are scheduled in a work-conserving manner. Through a CPU cap mechanism, it also ensures that no task starves for the CPU. Furthermore, it allows dynamic reconfiguration of the temporal partitions.
- It describes the design and implementation of the Secure Transport layer, which is a novel kernel-level communication mechanism for providing secure information flows between processes. It also ensures that at the kernel-level there are no blocking dependencies between tasks of different criticality levels.
- It empirically validates the design in the context of a case

study of a managed DRE system running on a laboratory testbed.

The outline of this paper is as follows: Section II presents the related research; Section III provides background information and the system model; Section IV presents the scheduler design; Section V describes the secure transport layer; Section VI empirically evaluates *DREMS* OS in the context of a representative application; and finally Section VII offers concluding remarks referring to future work.

II. Related Research

Our approach has been inspired by two areas: mixed criticality systems and partitioning operating systems. Mixed criticality systems provide support to multiple functionalities that can be of different criticality, or importance to the system. 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 avionic system can have safety-critical, mission-critical, and non-critical tasks.

In his seminal paper on mixed criticality scheduling, Vestal [5] argued that the criticality levels directly impact the task parameters, especially the worst-case execution time (WCET). In his framework, each task has a maximum criticality level and a non-increasing 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 level results in a more conservative verification process. Vestal [5] extended the response-time analysis of fixed priority scheduling to mixed criticality task sets. Vestal's results were later improved by Baruah et al. [6] where an implementation was proposed for fixed priority single processor scheduling of mixed-criticality tasks with optimal priority assignment and response-time analysis. A recent review on mixed-criticality systems research can be found in [7].

Partitioning operating systems provide applications 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 domain that 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 (CPU) resources during a dedicated time regardless of other applications.

Partitioning operating systems have been applied to avionics (e.g., LynxOS-178 [8]), automotive (e.g., Tresos, the operating system defined in AUTOSAR [9]), and cross-industry domains (DECOS OS [10]). A comparison of the mentioned partitioning operating systems can be found in [11].

Our approach combines mixed-criticality and partitioning techniques to meet the requirements of distributed real-time embedded systems where security plays a special role. *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.

Our work has many similarities with the resource-centric real-time kernel [12] 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 enforcing resource isolation among applications. However, to the best of our knowledge, [12] does not include support for process management, temporal isolation guarantees, partition management, and secure communication all at once.

III. Background and System Model

This section provides background material and states assumptions for the rest of the paper.

A. DREMS Architecture

DREMS [2] is a distributed system that consists of one or more computing nodes grouped into a cluster. We assume that at least one network route exists between any two nodes in the cluster. Distributed applications composed from cooperating processes called "actors" provide services for the end-user. Actors specialize the notion of OS processes; they have persistent identity that allows them to be transparently migrated between nodes, and they have strict limits on resources that they can use. Each actor is constructed from one or more reusable components [4] where each component is single-threaded.

B. The Linux Scheduler

The *DREMS* OS scheduler builts upon the standard Linux scheduler (kernel version: 3.2.7). The scheduler is responsible for allocating CPU resource(s) to all currently running computational entities. Schedulers are implemented in the Linux kernel through *scheduler classes*. The two important scheduler classes are CFS (Completely Fair Scheduler) and the RT (Real Time) scheduler [13]. The CFS scheduler attempts to allocate CPU time between processes fairly, while the RT scheduler selects processes based on their priority. Tasks eligible for scheduling are maintained in a structure called the *runqueue*.

A runqueue is a structure associated with each CPU. The runqueue is not necessarily just a queue – it is a container using a data structure that contains a bit array – one bit for each priority level and a list containing the tasks ready to be scheduled at that level. The bit at a level is set to one when there are tasks at that level. A 0 value indicates an empty queue at that level. In a multi core system, this structure is replicated per CPU. In a fully preemptive mode, the scheduling decision evaluates 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.

C. Task Levels and Temporal Partitioning

While most tasks[1] perform application functions, some tasks are used for system management and mission-critical functions. Thus, we group these tasks into different criticality levels: (a) *Critical* tasks are those tasks which are required

[1]We use the term threads and tasks interchangeably in this paper.

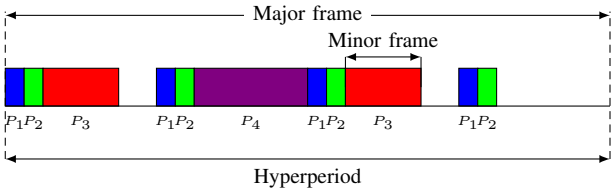


Fig. 1: A Major Frame. The four partitions (period,duration) in this frame are P_1 (2s, 0.25s), P_2 (2s, 0.25s), P_3 (4s, 1s), and P_4 (8s, 1.5s).

for system and mission management; (b) *Application* tasks perform mission-specific, non-critical work; (c) *Best Effort* tasks are those low priority tasks that are scheduled only when there are no runnable tasks from the previous two categories.

The system guarantees performance isolation between processes by (a) providing separate address spaces per actor; (b) enforcing that an I/O device can be accessed by only one actor at a time; and (c) facilitating temporal isolation between actors by the scheduler. Temporal isolation is provided via ARINC-653 [1] style partitions – a periodically repeating fixed interval of the CPU's time exclusively assigned to a group of cooperating actors of the same application.

A temporal partition is characterized by two parameters: period and duration. The period reflects how often the tasks within the partition will be guaranteed CPU allocation. The duration governs the length of the CPU allocation window in each cycle. Given the period and duration of all temporal partitions, an execution schedule can be generated by solving a series of constraints, see [14]. A feasible solution, *e.g.* Figure 1, comprises a repeating frame of windows, where each window is assigned to a partition. These windows are called *minor frames*. The length of a window assigned to a partition is always the same as the duration of that partition. The repeating frame of minor frames, known as the *major frame*, has a length called the *hyperperiod*. The hyperperiod is the lowest common multiple of the partition periods.

IV. *DREMS* OS SCHEDULER

A. Criticality Levels Supported by the DREMS OS Scheduler

The *DREMS* OS scheduler provides the ability to manage computation time for tasks at three different criticality levels: *Critical*, *Application* and *Best Effort*. The *Critical* tasks provide kernel level services and system management services. These task will be scheduled based on their priority whenever they are ready. *Application* tasks are mission specific and are isolated from each other. These tasks are constrained by temporal partitioning and can be preempted by tasks of the *Critical* level. Finally, *Best Effort* tasks are executed whenever no tasks of any higher criticality level are available.

Note that actors in an application can have different criticality levels, but all tasks associated with an actor must have the same criticality level, *i.e.* an actor cannot have both *Critical* tasks and *Application* tasks.

B. Modifications to the runqueue

To support the different levels of criticality, we extend the runqueue data structure described in Section III-B by creating one runqueue per partition per CPU. Currently, the system can support 64 temporal partitions. One runqueue is created for the critical tasks. The Best effort tasks are managed through the Linux Completely Fair Scheduler runqueue.

C. CPU Cap and Work Conserving Behavior

The schedulability of the *Application* level tasks is constrained by the current load coming from the *Critical* tasks and the temporal partitioning used on the *Application* level. Should the load of the *Critical* tasks exceed a threshold the system will not be able to schedule tasks on the *Application* level. A formal analysis of the response-time of the *Application* level tasks will not be provided in this paper, however, we present a description of the method we will use to address the analysis which will build on available results [6], [15], [16].

The submitted load function $H_i(t)$ determines the maximum load submitted to a partition by the task τ_i itself after its release together with all higher priority tasks belonging to the same partition. The availability function $A_S(t)$ returns for each time instant the cumulative computation time available for the partition[2] to execute tasks. The response-time of a task τ_i is the time when $H_i(t)$ intersects the availability function $A_S(t)$ for the first time. In our system $A_S(t)$ is decreased by the load of the available *Critical* tasks which if unbounded could block the application level tasks forever. This motivates us to enforce a bound on the load of the *Critical* tasks. This bound will be referred to as CPU cap.

In *DREMS* OS the CPU cap can be applied to tasks on the *Critical* and *Application* level to provide scheduling fairness within a partition or hyperperiod. Between criticality levels, the CPU cap provides the ability to prevent higher criticality tasks from starving lower criticality tasks of the CPU. On the *Application* level, the CPU cap can be used to bound the CPU consumption of higher priority tasks to allow the execution of lower priority tasks inside the same partition. If 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 enforcement is performed in 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 CPU cap is reached, the task is not marked ready for execution unless (a) there is no other ready task in the system; or (b) the CPU cap accounting is reset. This behavior ensures that the kernel tasks such as those belonging to Secure Transport, discussed in Section V, 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 (a) there are no *Critical* tasks to schedule and there are no other ready tasks in the partition; or (b) the CPU cap accounting has been reset.

[2]In the original model [15] $A_S(t)$ is the availability function of a periodic server.

TABLE I: *DREMS* Symbols used in Section IV

APP_INACTIVE	The scheduler state in which tasks in temporal partitions are not scheduled
APP_ACTIVE	Inverse of APP_INACTIVE
$firstrun$	A global variable, set whenever the major frame has been changed
mfl	A global circular linked list of minor frames used by the scheduler
cur_frame	Current minor frame.
HP_start	Global variable, stores the start time of a new major frame.

Procedure 1 Update Major frame

Input: $frame$ {A sorted but not necessarily contiguous major frame structure}
Input: $Valid(mf)$
1: $Reassign\ Task\ to\ CPU\ 0$
2: $Acquire\ update\ frame\ spinlock,\ disable\ preemption/interrupts$
3: $frame \leftarrow Fill\_Empty(frame)$
4: **Atomic** $:state \leftarrow APP\_INACTIVE$
5: $firstrun \leftarrow true$
6: $mfl \leftarrow frame.minorframelist$
7: **Atomic** $:state \leftarrow APP\_ACTIVE$
8: $Release\ update\ frame\ spinlock,\ enable\ preemption/interrupts$

D. Major Frame Configuration

This section describes the mechanism used to configure (or reconfigure during a mission) the partition scheduler, Procedure 1. Table I summarizes the key symbols used in this and related subsections. During the configuration process that can be repeated at any time without rebooting the node, the kernel receives a major frame structure that contains a list of minor frames. It also contains the length of the hyperperiod, partition periodicity and duration. Note that major frame reconfiguration can only be performed by a process with suitable capabilities. The *DREMS* capability model is not discussed in this paper, it can be found in [2].

Before the frames are set, 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: $(\forall p \in \mathbb{P})(O^p_1 \leq \phi(p))$; (C2) Time between any two executions should be equal to the partition period: $(\forall p \in \mathbb{P})(k \in [1, N(p)-1])(O^p_{k+1} = O^p_k + \phi(p))$., 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^p_i is the offset of i^{th} minor frame for partition p from the start of the major frame, H is the hyper period.

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

If the constraints are satisfied, then the task is moved to *CPU0* if it is not already on *CPU0*. This is done because the global tick (explained in next subsection) 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`, 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 2.

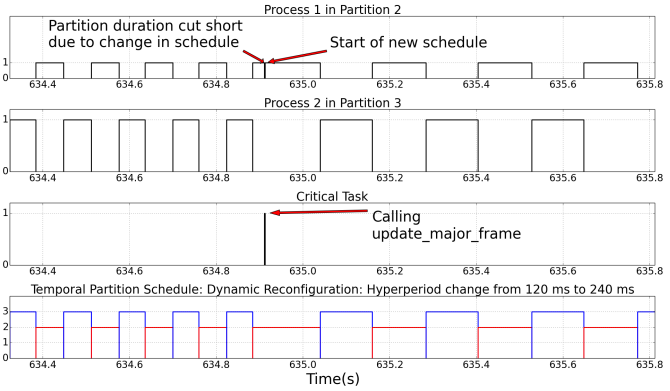


Fig. 2: Two single-threaded processes run in separate partitions with a duration of $60ms$ each. The schedule is dynamically reconfigured so that each partition duration is doubled. A *Critical* task is responsible for calling the update_major_frame system call. Duration of the active partition is cut short at the point when update_major_frame function is called.

Note that, though it is not shown in the algorithm, it is possible to set the global tick to be started with a delay. 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.

E. Main Scheduling Loop

A periodic tick running at 250 Hz[3] 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 Procedure 2 in interrupt context only on *CPU0*. Procedure 2 is executed only *CPU0* so that every CPU switches the current partition at approximately the same time, to within one global tick of the scheduler. This algorithm enforces the partition scheduling and updates the current minor frame and hyperperiod start time (`HP_start`). The partition schedule is determined by the mfl, which is a circular linked list of minor frames which comprise the major frame. Each entry in the mfl 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 main scheduler function, described in Procedure 3, executes. This scheduler function is run on each processor. The main part of the scheduler function is to get the next runnable

[3]The kernel tick value is also called 'jiffy' and can be set to a different value when the kernel image is being compiled

Procedure 2 Global Tick

```
1: if {Current CPU is CPU0} then
2:     if firstrun and mfl ≠ null then
3:         firstrun ← false
4:         HP_start ← Sched_clock(){ Sched_clock() provides the current uptime measured based on elapsed jiffies}
5:         MF_start ← HP_start
6:         cur_frame ← HEAD(mfl)
7:         next_switch ← HP_start + cur_frame.duration
8:     end if
9:     if Sched_clock() ≥ next_switch then
10:        cur_frame ← cur_frame.next
11:        next_switch ← next_switch + cur_frame.duration
12:        if cur_frame == HEAD(mfl) then
13:            HP_start ← Sched_clock()
14:        end if
15:    end if
16: end if
```

Procedure 3 Main Scheduler Function - Called when task wishes to give up the CPU or a CPU tick occurs

```
Input: TIF_NEED_RESCHED flag on the task is set by scheduler_tick(). Preemption is enabled.
1: Disable(Preemption)
2: RQ ← Get_CPU_RQ(CurrentCPU)
3: prev_task ← RQ.curr_task
4: [index, next_task] ← Pick_Next_Task(RQ, sys_partition)
5: if index >= MAX_RT_PRIO and state ≠ APP_INACTIVE then
6:     [index, next_task] ← Pick_Next_Task(RQ, cur_frame.partition)
7:     if index >= MAX_RT_PRIO then
8:         [index, next_task] ← Pick_Next_Best_Effort_Task()
9:     end if
10: end if
11: Update_Exec_Time(prev_task)
12: RQ.curr_task ← next_task
13: if {CPU Cap Enabled} then
14:     Update_Stats(prev_task)
15:     Update_Disabled_Bit(prev_task)
16: end if
17: if prev_task! = next_task then
18:     Context_Switch{RQ, prev_task, next_task}
19: end if
20: Enable(Preemption)
```

task from the runqueues, corresponding to lines $4-10$ in the algorithm. These lines show the implementation of the *mixed criticality* scheduling into 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 criticality is displayed in the algorithm, as *Pick_Next_Task* is called first for the system partition. Only if there are no runnable *Critical* system tasks and the scheduler state is not set to $APP\_INACTIVE$ 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.

F. Pick Next Task and CPU Cap

The *Pick_Next_Task* algorithm, described in Procedure 4, returns MAX_RT_PRIO and $null$ if there are no runnable tasks in the runqueue. If CPU cap is disabled, the *Pick_Next_Task* algorithm returns the first task from the specified runqueue (see Section IV-B). The Pick_Next_Best_Effort_Task() algorithm is not shown as it is the default algorithm for the Completely Fair Scheduler class, as implemented in Linux Kernel [13].

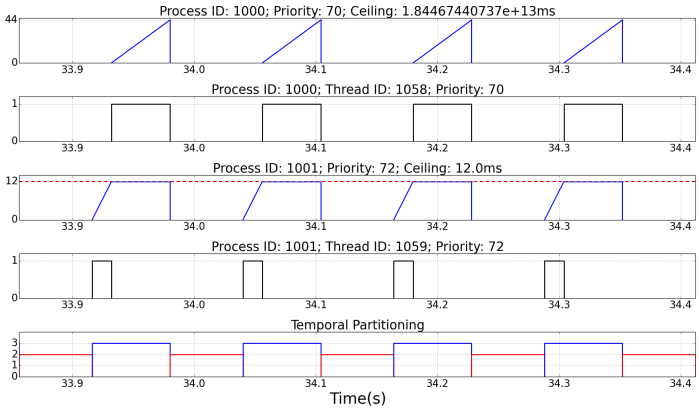


Fig. 3: Single Threaded processes 1000 and 1001 share a partition with a duration of $60ms$. Process 1000 has 100% CPU cap and priority 70; process 1001 has 20% CPU cap, and higher priority 72. Since process 1001 has a CPU cap less than 100%, a ceiling is calculated for this process: 20% of $60ms$ = $12ms$. The average jitter was calculated to be 2.136 ms with a maximum jitter of 4.0001 ms.

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 as it 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 reenables 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. If the current runqueue belongs to the system or critical partition then MAX_RT_PRIO and $null$ since the CPU cap for critical tasks is a hard limit; see section IV-C for a discussion of this behavior.

This iteration through the task list when CPU cap enforcement is enabled increases the complexity of this algorithm to $O(n)$, where n is the number of tasks in that temporal partition, from the Linux scheduler's complexity of $O(1)$. Note that this complexity increase occurs only when CPU cap enforcement is enabled and there is at least one actor that has less than 100% CPU cap. In the worst case, all actors are given a partial (less than 100%) 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, as seen in lines $13-16$ of Procedure 3.

Example: Figure 3, shows the above mentioned scheduler decisions when CPU cap is placed on processes that share a temporal partition. To facilitate analysis, the scheduler uses a logging framework that updates a log every time a context switch happens. Figure 3 clearly shows the lower priority actor executing after the higher priority actor has reached its CPU cap. If the CPU cap of the higher priority process is set to 100%, it has no ceiling and therefore consumes all of the CPU

Procedure 4 Pick Next Task from RunQueue

Input: RQ {The scheduler runqueue}; $partition$ {The currently active partition}

```
prio_array ← RQ.PartitionRQ[partition]
next_task ← null
next_index ← MAX_RT_PRIO
index ← 0
while index < MAX_RT_PRIO do
    index ← FindFirstBit(prio_array.bitmap + index) {find first enabled bit after index}
    if index >= MAX_RT_PRIO then
        return MAX_RT_PRIO, null
    end if
    runlist ← prio_array.queue + index {runlist is a doubly linked list containing all the tasks at that priority level}
    if next_task == null then
        next_task ← runlist[0]
        next_index ← index
    end if
    if {CPU Cap Enabled} then
        for task in runlist do
            if task.disabled == true then
                if task.last_disabled_time < CPUCAP_WIN_start then
                    task.disabled ← false
                    next_task ← task
                    next_index ← index
                    return next_index, next_task
                end if
            else
                next_task ← task
                next_index ← index
                return next_index, next_task
            end if
        end for
    else
        return next_index, next_task{CPU CAP is DISABLED}
    end if
end while{This implies that all tasks are disabled due to CPU cap.}
if partition == sys_partition then
    return MAX_RT_PRIO, null {the CPU cap for critical tasks is a hard limit.}
end if
return next_index, next_task {returns the highest priority task if all tasks are disabled or returns a null task with MAX_RT_PRIO.}
```

time within the partition (not shown in the figure). From the scheduler log, the average jitter was calculated to be 2.136 ms with a maximum jitter of 4.0001 ms. This jitter is consistent with the value of jiffy used, 4 ms, indicating that the scheduler would occasionally take an extra jiffy of time to switch to the next minor frame. Also, no overshoot was observed in the thread activity for all processes - neither process executed outside its temporal partition.

V. Secure Transport

In order to support communication and coordination between applications of different criticality, priority, and security levels, we developed the DREMS 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 Multi-Level Security (MLS) policy ([17], [18], [2]), both of which must be configured for each task by a trusted system administration infrastructure. 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.

The remainder of this section will provide an overview of the core concepts of DREMS Secure Transport and discuss the architecture of its implementation. Discussion of the MLS policy and its enforcement or detailed empirical evaluation of this facility, however, is outside the scope of this paper and will not be included.

A. Secure Transport Overview

1) Endpoints: Endpoints are the basic communication artifact used by applications to transmit and receive messages; they are analogous to socket handles in traditional BSD socket APIs. 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 inter-process communication with other tasks; such endpoints must be explicitly configured by a privileged task acting as a trusted system configuration infrastructure.

Endpoints are separated into four different categories with different restrictions on their creation and use; for the purposes of this discussion, we will describe only the two endpoint classes that are used for IPC:

- **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.
- **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 through a network.

2) 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 is allowed to 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 a inbound flow from the sender.

B. Secure Transport Architecture

The implementation of Secure Transport in DREMS is broadly divided into two halves, which we will call the "ST Support" and the "ST Reactor". The Support portion of the ST implementation contains most of the support infrastructure, *i.e.* bookkeeping data and business logic required to implement

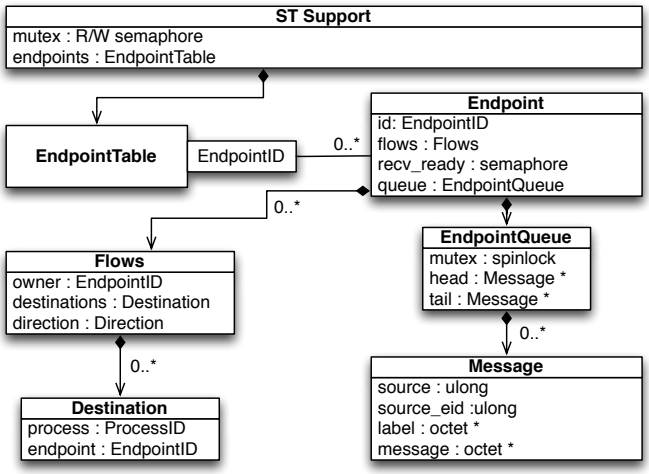


Fig. 4: Simplified ST Support Architecture

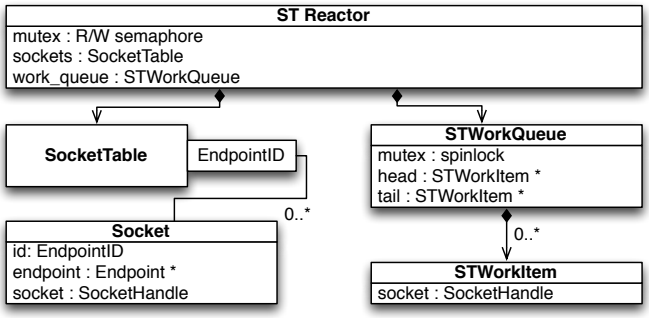


Fig. 5: Simplified ST Reactor Architecture

basic operation, including enforcement of flows and security restrictions. This portion also serves as the primary point of entry for all system calls related to ST. The Reactor contains the data structures and business logic required to send and receive remote messages using the underlying UDP and SCTP protocol stacks.

1) Secure Transport Support: Access to ST Support is controlled by a single read/write mutex preventing race conditions that may occur when a privileged task attempts to update the system endpoint/flow configuration while regular tasks are performing unprivileged operations. This read/write mutex has three important properties:

1) It ensures that when a write lock is requested, no further read locks are granted; ensuring that a pending write (typically by a high priority task) will not be indefinitely blocked.
2) It allows multiple readers to acquire the lock at once, reducing the possibility of a priority inversion blocking access to high priority tasks.
3) It is a *sleeping lock*, meaning that if the mutex cannot be acquired by a task, it is marked inactive and not scheduled again until either a timeout occurs or the lock can be obtained.
4) It prevents *priority inversion* by supporting priority inheritance: when a high priority process is blocked while trying to acquire the mutex, the priority of the current owner(s) is boosted to that of the top priority waiter until they release the mutex.

A *write* lock is obtained whenever a system call is entered that would create, update, or modify an endpoint - such system calls may only be used by high priority privileged tasks. Regular system calls — *e.g.* send, receive, or select — acquire the mutex with a *read* lock only. This allows multiple tasks of different criticality and priority levels to be active in the ST infrastructure at once.

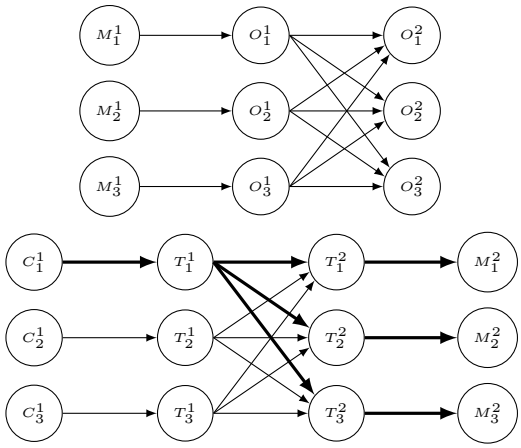
Most data structures are satisfactorily protected using this scheme: all data structures (for example, tables maintaining endpoint configuration, flow configuration, and destination mappings) with one exception (discussed below) are read-only to tasks performing unprivileged ST operations.

For this reason, most of the design of the ST Support is outside the scope of this paper. The data structures that maintain state for endpoints, shown in Figure 4, however, merit further discussion. Endpoints are represented in the kernel by a data structure that contains:

- Basic properties of the endpoint, *e.g.* the identifier, type, security label, and maximum allowed buffer sizes.
- The endpoint queue, a linked list of message objects waiting to be received.
- A spinlock used to synchronize access to the endpoint queue.
- A mutex (sleeping lock) used to wait for receipt of messages.

The basic properties of the endpoint are not modified by tasks accessing the endpoint through non-privileged system calls, and are sufficiently protected by the ST read-write mutex described above. The endpoint queue, however, is expected to be manipulated by these non-privileged system calls, and if not properly protected could be a source of priority inversions and deadlocks. In order to avoid these cases, the endpoint operations use a spinlock - a low overhead busy wait mutex that temporarily disables preemption while in its critical section, to protect access to the endpoint queue. In the event that a task attempts to receive a message from an empty queue, a mutex (sleeping lock) is available so the task can yield to lower priority tasks that may be attempting to send messages into that queue. In order to illustrate this, we will describe the process through which messages are sent and received for LMEs.

a) Local Message Endpoint - Sending a Message: When a task enters the *send* system call, it first acquires a read lock on the ST mutex. After successfully acquiring the mutex, it retrieves the endpoint structure for the sending endpoint and checks for a valid outbound flow for the destination. Presuming a valid flow is found, it then retrieves the endpoint structure for the destination endpoint, and checks that a valid destination flow exists. If both flow checks pass, the task constructs a message object to hold the message. The task then acquires the destination endpoint's spinlock for only as long as required to place the message object on the endpoint queue's linked list, and signals the endpoint mutex before releasing the spinlock. Note that when the spinlock is acquired, the task cannot be preempted until the spinlock is released.



Task	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

Fig. 6: *DREMS* tasks : *ModuleProxy* tasks control thruster activation in Orbiter and state vector retrieval from Orbiter. *OrbitalMantenance* tasks track the cluster satellites' state vectors and disseminate them. *TrajectoryPlanning* tasks control the response to commands and satellite thruster activation. *CommandProxy* tasks inform the satellite of a command from the ground network. For these tasks, the subscript represents the node ID on which the task is deployed. The total latency of the interaction $C_1^1 \rightarrow M_N^2$ represents the total emergency response latency between receiving the *scatter* command and activating the thrusters. This interaction pathway is bolded.

b) Local Message Endpoint - Receiving a Message: When a task enters the *receive* system call, it first acquires a read lock on the ST mutex. After successfully acquiring the mutex, it retrieves the endpoint structure for the receiving endpoint. The task acquires the spinlock on this endpoint structure and attempts to remove a message from the linked list. If no message exists, it releases the spinlock and performs a timed sleep on the mutex. Once awakened (either through a signal from a sender or through timeout), it re-acquires the spinlock and attempts again to dequeue a message.

c) Synchronization Analysis: In the tasks outlined above, two tasks of different priority levels can execute most of the tasks of sending or receiving a message without contending for a mutex. Most of the computationally intensive work required for sending or receiving a message happens while holding only the ST R/W mutex and executes at their respective priority levels. The only place where such tasks may contend is when inserting or removing an item in a linked list, a very fast O(1) operation. Since that preemption is disabled once the lock is acquired, we ensure that a higher priority task cannot preempt this operation and prevent it from completing.

2) Secure Transport Reactor: The Reactor portion of the ST architecture, a simplified version of which is shown in Figure 5, consists of the data structures and business logic required to send messages to tasks on other nodes. Similar to the ST Support, the ST Reactor has a read/write mutex to synchronize access to its internal data structures. Unlike the ST Support, however, there is no requirement for per-endpoint synchronization. Also unlike the ST Support the ST Reactor is not entirely driven by threads from user level tasks. The ST Reactor maintains a pool of kernel threads that handle receipt of messages from the network stack and deliver messages to their destination endpoint queues. To illustrate this, we will describe sending and receiving a message on RMEs.

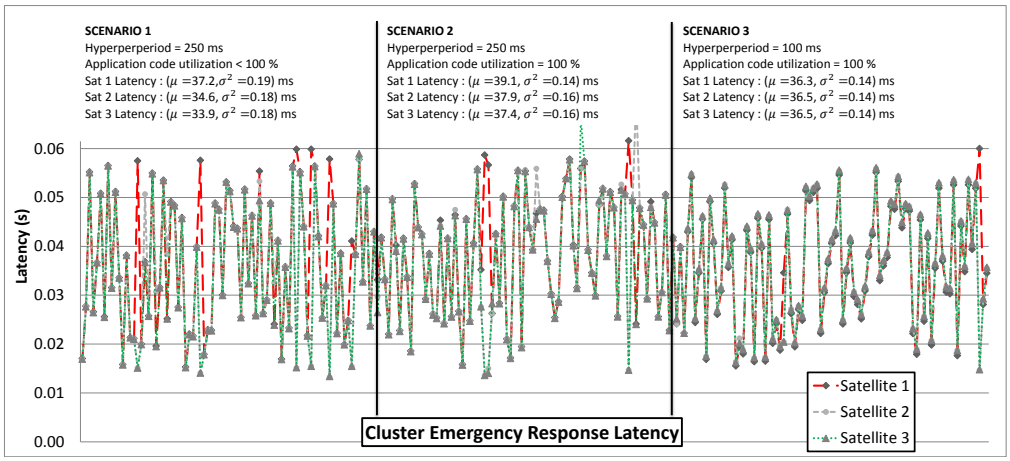
a) Remote Message Endpoint - Sending a Message: Sending a message on RME is similar to sending a message on a LME until the sending side flow check has succeeded. At that point the task constructs the message object and enters the ST Reactor. Then the task acquires a read lock on the ST Reactor mutex and retrieves the necessary data structure (*e.g.* socket handle) needed to send a message, using non-blocking semantics, to the desired destination address. Finally the task releases both the Support and Reactor mutexs and attempts to send the message.

b) Remote Message Endpoint - Receiving a Message: When a message arrives from the network stack, the ST Reactor is notified via callback that a message has arrived. When this occurs, an item containing the socket handle is enqueued on a work queue serviced by the ST Reactor thread pool. When one of the members of this thread pool accepts the work item, it allocates a buffer and retrieves the data from the network stack. This buffer is then immediately delievered onto the endpoint queue of the recipient endpoint (requiring only that the reactor thread hold the queue-specific spinlock) and wakes any processes that may be blocked on `receive` or `select`. When a process elects to receive the message, the message is decoded and flows and security labels are checked in the context of that process, using its default priority.

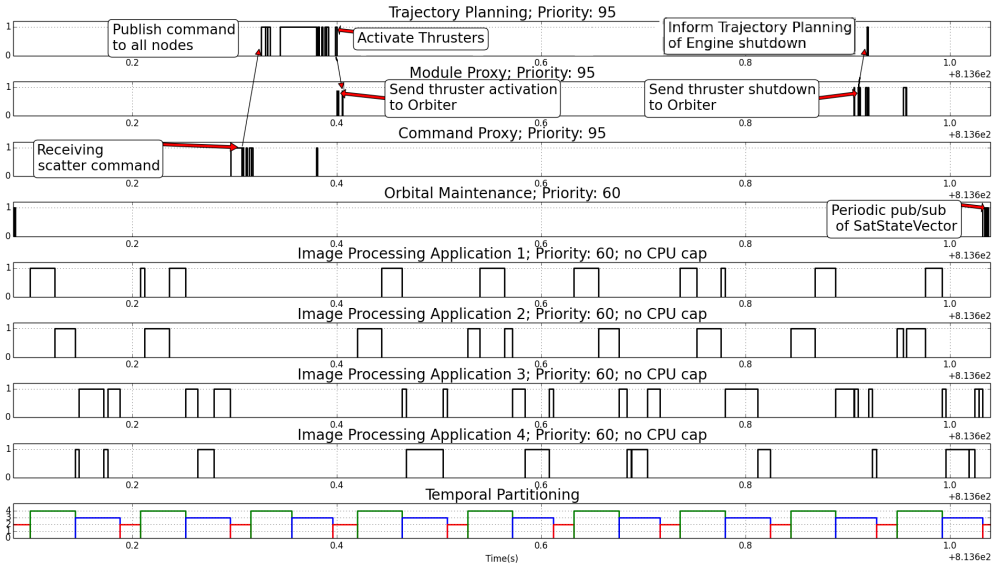
c) Synchronization Analysis: Tasks sending messages need only contend for read locks on both the Support and Reactor portions of the ST infrastructure. Therefore such tasks will not interfere with other higher priority tasks trying to send or receive. Receipt of messages by the Reactor kernel threads merits some further discussion: the priority and scope of these threads must be carefully chosen so as to not interfere with the operation of any critical tasks. Several tasks must be accomplished to correctly deliver messages: buffers must be allocated to hold the message content, metadata attached to the message must be demarshalled, finally flow and label checks must be conducted. Performing all of these tasks early (*i.e.* in the scope of the reactor thread pool) ensures that kernel memory is not wasted on messages that might fail any one of these stages and not be delivered. If these threads are scheduled as real-time tasks, high levels of network traffic, *e.g.* a denial of service attack, would starve any tasks of lower priority. For this reason, we have elected to defer most of these activities to actual receipt of a message by a process: the reactor thread simply allocates a buffer to hold the message and enqueues it on the appropriate endpoint queue. Thus, it is safer to run these threads at maximum priority, and the computationaly intensive parts of message delivery run at the actual process priority when they are received.

VI. Experiment: A 3-node Cluster

To demonstrate the DREMS platform, a multi-computing node experiment was created on a cluster of fanless computing nodes with a 1.6 GHz Atom N270 processor and 1 GB of RAM each. On these nodes, a cluster of three satellites was emulated



(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 6. 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 shown above. The *scatter* command causes the *TrajectoryPlanning* to request *ModuleProxy* to activate the thrusters for 500 ms. Notice that the image processing does not run while the mission-critical tasks are executing - without halting the partition scheduling. Also note that the context switching during the execution of the critical tasks is the execution of the secure transport kernel thread. Only the application tasks are shown in the log; the kernel threads and other background processes are left out for clarity.

Fig. 7: *DREMS* Mixed Criticality Demo

and each satellite ran the example applications described in Section I. We use these example applications to show that the performance of mission-critical tasks is not affected by application tasks. Because the performance of the cluster flight control application is of interest, we explain the interactions between its actors below.

The mission-critical cluster flight application (CFA) 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 group publish subcribe interaction between all Orbital Maintenance actors across all nodes. These CFA tasks and their interactions are further explained in Figure 6.

Additionally, four image processing application (IPA) actors (1 actor per application instance) are deployed as application tasks. These IPAs were written so that we can configure the percentage of CPU cycles consumed by them. The four IPAs are assigned to two partitions, such that each partition contains two IPA actors. There is a third, shorter, partition in which runs the *OrbitalMaintenance* actor, since it is a periodic task - it updates the satellite state every second - and is not

critical in an emergency.

To ensure the cluster safety, the latency between the reception of the command from the ground station and the activation of the satellites' thrusters should be minimized, *i.e.* regardless of other applications, the latency of the interaction $C_1^1 \rightarrow M_N^2$, described in Figure 6, should remain as small as possible. Note: since only the cluster leader (node *1*) receives the scatter command, *CommandProxy* and command publish on nodes *2* and *3* are not active. These latencies were calculated from time-stamped messages. Using NTP (http://www.ntp.org), all nodes' clocks were synchronized to within 10 μs.

Figures 7a and 7b 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. As shown in figure 7a, 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. Since we show that the emergency response has very low latency with little variance between different application loads on the system, we provide a stable platform for deterministic and reliable emergency response. As such, the satellite cluster running the *DREMS* infrastructure is able to quickly respond to emergency situations despite high application CPU load and without altering the partition scheduling. Figure 7b and demonstrates the proper preemption of the image processing tasks by the critical CFA tasks for scenario 2.

VII. Conclusions and Future Work

This paper propounds the notion of managed distributed real-time and embedded (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. The timing constraints and resource sharing require effective mechanisms that can assure both performance isolation and secure communications for their correct application operation. To address these requirements, this paper describes the design and implementation of a distributed operating system called *DREMS* OS focusing on two key mechanisms: the scheduler and secure transport mechanism that work synergistically. *DREMS* OS is part of a larger project comprising both design-time and run-time tools.

We have verified the behavioral properties of the OS scheduler, focusing on temporal and spatial process isolation, safe operation with mixed criticality, precise control of process CPU utilization and dynamic partition schedule reconfiguration. We have also analyzed the scheduler and network level properties of a distributed application built entirely using this platform and hosted on an emulated cluster of satellites showcasing the usefulness of model-driven design-time tools with support for model checking and code generation significantly reducing the complexity for the developer.

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 scheduling the component's operations. Additionally, such complex networked systems with mission-critical tasks distributed among many nodes require guarantees about the network Quality of Service (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 the DARPA System F6 Program under contract NNA11AC08C. 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. The authors thank Olin Sibert of Oxford Systems and all the team members of our project for their invaluable input and contributions to this effort.

References

[1] *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, ARINC Incorporated, Annapolis, Maryland, USA, Jan. 1997.

[2] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose, "A Software Platform for Fractionated Spacecraft," in *Proceedings of the IEEE Aerospace Conference, 2012*. Big Sky, MT, USA: IEEE, Mar. 2012, pp. 1–20.

[3] A. Dubey, A. Gokhale, G. Karsai, W. Otte, and J. Willemsen, "A Model-Driven Software Component Framework for Fractionated Spacecraft," in *Proceedings of the 5th International Conference on Spacecraft Formation Flying Missions and Technologies (SFFMT)*. Munich, Germany: IEEE, May 2013.

[4] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in *Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13)*, Paderborn, Germany, Jun. 2013.

[5] S. Vestal, "Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance," in *Proc. of 28th IEEE Real-Time Systems Symposium*, Tucson, AZ, Dec. 2007, pp. 239–243.

[6] S. Baruah, A. Burns, and R. Davis, "Response-Time Analysis for Mixed-Criticality Systems," in *Proceedings of the 2011 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov. 2011, pp. 34–43.

[7] A. Burns and R. Davis, "Mixed Criticality Systems – A Review," Department of Computer Science, University of York, Tech. Rep., 2013. [Online]. Available: http://www-users.cs.york.ac.uk/~burns/review.pdf

[8] LynuxWorks, "RTOS for Software Certification: LynxOS-178." [Online]. Available: http://www.lynuxworks.com/rtos/rtos-178.php

[9] Autosar GbR, "AUTomotive Open System ARchitecture." [Online]. Available: http://www.autosar.org/

[10] R. Obermaisser, P. Peti, B. Huber, and C. E. Salloum, "DECOS: An Integrated Time-Triggered Architecture," *e&i journal (Journal of the Austrian Professional Institution for Electrical and Information Engineering)*, vol. 123, no. 3, pp. 83–95, Mar. 2006.

[11] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, "A Comparison of Partitioning Operating Systems for Integrated Systems," in *Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4680/2007, pp. 342–355.

[12] K. Lakshmanan and R. Rajkumar, "Distributed Resource Kernels: OS Suppport for End-To-End Resource Isolation ," in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, Apr. 2008, pp. 195–204.

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

[14] A. Dubey, G. Karsai, and N. Mahadevan, "A Component Model for Hard Real-time Systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517–1550, 2011.

[15] L. Almeida and P. Pedreiras, "Scheduling within Temporal Partitions: Response-time Analysis and Server Design," in *Proc. of the 4th ACM Int Conf on Embedded Software*, Pisa, Italy, Sep. 2004, pp. 95–103.

[16] G. Lipari and E. Bini, "A Methodology for Designing Hierarchical Scheduling Systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, Apr. 2005.

[17] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE, Technical Report 2547, Volume I, 1973.

[18] O. Sibert, "Multiple-domain labels," 2011, presented at the F6 Security Kickoff.