Software Organization to Facilitate Dynamic Processor Scheduling

Raymond K. Clark
The MITRE Corporation
Bedford, MA 01730
rkc@mitre.org

E. Douglas Jensen
The MITRE Corporation
Bedford, MA 01730
jensen@real-time.org

Nicolas F. Rouquette
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
nicolas.rouquette@jpl.nasa.gov

Abstract

The NASA Jet Propulsion Laboratory (JPL) is developing the Mission Data System (MDS) for potential use in future space missions, where it is expected to reduce the complexity and effort required to produce mission and ground-support software, while also enabling greater autonomy for remotely deployed systems, including future planetary rovers. The MDS software architecture includes two levels of processor scheduling: a high-level planner and a low-level execution manager. JPL and MITRE are investigating the use of Time-Utility Functions to manage low-level scheduling, since they promise to enable adaptive, short-term processor scheduling based on mission utility. Recent development work on MDS has focused on organizing the software so that low-level processor scheduling will be most effective. This paper describes the major organizing principles that have shaped this work.

1. Introduction

Future space missions are becoming more challenging and complex. Flybys of planets are now being followed with landers and in-situ mobile vehicles. Mobile vehicles have demanding requirements to achieve scientific objectives without the intervention of ground controllers. Obstacles must be avoided. Objects of interest need to be efficiently identified and investigated on the way to achieving specific ground-directed mission objectives. Timely communications with Earth may be impractical because of the long distances involved, or because of blocked communications. Missions that previously could be planned in advance as detailed sequences of commands, now must be flexible to handle uncertainties in time, trajectory, and science opportunities.

The availability of more capable and affordable computing power enables the above mission objectives. But it also puts more responsibility on software to implement more advanced capabilities.

The Mission Data System [10, 2] (MDS) is a unified flight, ground, and test architecture under development at the Jet Propulsion Laboratory that will address these problems. It will provide a set of reusable software frameworks, inspectable models, and a methodology and language to identify and express requirements in their terms.

MITRE and JPL are exploring the use of time-utility functions (TUFs) to schedule the processors for MDS. JPL feels that TUFs can play a key role in MDS by providing the flexible and responsive processor scheduling that is needed for the more complex, more autonomous missions of the future.

Using this approach, each task comprising the MDS workload will be characterized by a TUF, which specifies the utility to the overall system of completing that task as a function of time. A TUF scheduler will then apply heuristics that attempt to maximize overall accrued utility for the mission.

To date, several significant changes have been made in the MDS infrastructure to make it more amenable for dynamic scheduling, in general, and TUF scheduling, in particular. The code to gather the data that will ultimately be used to schedule the processor(s) using TUF had been designed and inserted into the MDS mainline codebase in preparation for experiments with TUF technology.

2. The MDS Architecture

The Mission Data System guiding belief is that software plays a central and increasingly important system role that must be reconciled with the overall systems engineering approach adopted by a project. Both software and systems engineering apply across all parts of a project and to all elements of the environment affecting the mission. Therefore, it is essential that systems and software share a common approach to defining, describing, developing, testing, operating, and visualizing what systems do. To realize this belief,

MDS is founded on a set of architectural themes that shape the design, and which are described below.

State. MDS is a state-based architecture, where state is a representation of the condition of an evolving system. All clients access state in a uniform way through *state variables*, as opposed to a program's local variables.

A state timeline describes a state's value as a function of time. State timelines are a complete record of the system's history, expectations, and plans. As well as providing the fundamental coordinating mechanism on the spacecraft, state timelines are also the objects of a uniform mechanism of information exchange between flight and ground.

An *estimator* weighs evidence from measurements, commands, and other states to calculate an estimate for a state's value. A state estimate is the system's best guess of the "true" physical state. MDS recognizes that state estimates are not "truth" and requires that all state estimates include an assessment of uncertainty.

Controllers are responsible for utilizing actuators in order to influence a state, as reflected in a state variable's value. They employ models (see below) to determine the actions required to modify in a desired manner.

Estimators and controllers are sometimes referred to collectively as *achievers*.

Models. MDS tries to express domain knowledge explicitly in inspectable *models* rather than implicitly in program logic. The models, which describe how states evolve and are affected, separate the application-specific knowledge from the reusable logic that applies domain knowledge to solve a problem. A model built for a specific mission may also be reused on future projects. The task of customizing MDS for a mission, then, becomes largely a task of defining and validating new or reused models.

Goals. All of the mission processing performed by MDS result from the explicit declaration of *goals*. A goal specifies a desired state that the system should attain. In addition, each goal has an associated *goal priority* that indicates its importance with respect to other goals.

The goals that are explicitly given to MDS can, in general, be satisfied in a number different ways. A planner can decompose a goal into a set of subgoals, which can be further decomposed, until finally a set of executable goals—constraints placed on a single state variable—have been derived.

Each state variable is monitored by an estimator and influenced by one or more controllers.

Software Architecture. Unfortunately, recent advances in software architecture and component technology, when applied to space missions, have occasionally resulted in software mistakes that have led projects to fail and missions to be lost. These missions have often been characterized by unreasonable levels of risk, as well as cost and schedule overruns.

To better capture the structure of a system and to enforce constraints that prevent certain classes of errors, the Mission Data System project advocates expressing software organization within a component/connector-based architecture. In its current form, the component/connector style used in MDS is designed to provide a simple strategy for software organization: components encapsulate the functional aspects of the system while connectors encapsulate the coordination aspects of the system. Explicit links among specific ports on components and connectors define how functional and coordination aspects are woven together.

Using a software architecture methodology goes well beyond the mere adoption of component-oriented programming and technology; it is also related to verification and validation. In the Mission Data System, the component/connector architecture style provides the initial basis with which to extract the software abstractions that are necessary to apply model checking effectively (because the scope of the model is small due to abstraction) and successfully (because the abstraction model is consistent with the implemented design).

2.1. Processor Scheduling within MDS

MDS scheduling happens at two distinct levels in the system, which has a great impact on the scope of our TUF-related work. There is a high-level planner named MPE (Mission Planning and Execution), which is an artificial intelligence-style planner that takes into account all required resources to accomplish tasks. (However, it might have only a very rough idea of the actual required execution time for a given task.) MPE produces partially ordered lists of flexible timepoints, each of which is characterized by the set of tasks that either begin or end at that timepoint. Timepoints are typically associated with the satisfaction of certain state constraints, rather than with the passage of a specific length of time. (While there might often be a known relationship between the passage of time and the change of state of a given state variable, this relationship is not always completely understood or might necessarily depend on aspects of the environment that are not entirely known—such as the obstacles present along a desired navigation path or the composition of the surface or the presence of specific characteristics of interest along a path.) Timepoints are only partially ordered so that maximum flexibility in goal satisfaction is possible.

The lower-level scheduler is responsible for scheduling the executable goals that the planner releases at specific timepoints. Before the notion of TUF was explored, this was a priority scheduler, where the priorities were specified a priori by the MDS application (or infrastructure) designers. This is the level where TUF scheduling technology is to be applied in the short term.

3. Time-Utility Function Scheduling

The previous discussion has focused on TUFs and TUF scheduling without elaborating on the precise definition of these terms. The term "TUF" has deliberately been used to capture the general notion that real-time computations must satisfy certain timing requirements and that their successful completion results in some utility to the mission.

Time-utility functions are the logical descendents of time-value functions [6]. Each computation in a real-time application is composed of a sequence of computational phases. The application as a whole makes progress when its component computations make progress; and each computation makes progress by completing its computational phases. Therefore, the completion of a computational phase marks measurable progress for the application, and this progress is expressed in terms of utility units. Associated with each phase, then, is a Time-Utility Function that specifies that phase's time constraint and utility.

A TUF is a mathematical function that describes the utility to the mission of completing a specific computation as a function of time. The mission utility is specified in application-defined units and can be drawn from a much larger space than, say, a typical priority space. Consequently, a higher utility value not only indicates that one computation is more valuable to the mission than another—it also indicates how much more important it is in mission-specific terms.

A TUF scheduler produces a tentative schedule based on the TUFs associated with all of the currently active computations in the system, as well as their estimates about the computational resources required to complete each computation. This schedule is produced by means of heuristics. The policies that have been produced thus far, including [7, 4], have behaved, when the system is not overloaded, like Earliest Deadline First schedulers and, when the system is overloaded, they have shed computations based on the utility-per-unit-time that they produce. These schedulers have accrued utility for their applications by adding together the utilities obtained from the successful completion of individual computations. (Many variations on these schedulers are possible and several are being pursued.)

There have been two implementations of TUF schedulers to date: the first was done for the Alpha Operating System [9] and the second was incorporated into The Open Group's MK7.3a Operating System [1]. These implementations have been used to demonstrate TUF (and other) technology in realistic application contexts—specifically, a coastal defense scenario [8] and an airborne tracking system [3].

3.1. The Role of Dynamic Scheduling in MDS

Given a plan produced by MPE, the low-level scheduler manages processor cycles to maintain the active constraints on designated state variables. Underlying the high-level, state-based model, there is a set of runtime components that communicate by means of stylized connectors and embody the implementation of the various achievers (i.e., estimators and controllers). In addition, there is a specification that binds a specific set of operating system threads to specific achievers. In the end, the low-level scheduler determines the order in which those threads are executed.

Clearly, if static priorities were associated with each if these threads, a priority scheduler could dispatch them in an acceptable order using common real-time technology. However, such a scheduler does not provide all of the capabilities that are desired for MDS. This section outlines some of the capabilities that are available—and useful—to MDS when a more dynamic policy is employed for low-level scheduling. Often, the description will focus explicitly on TUF scheduling; but it should be understood that much of the discussion might also apply to other dynamic scheduling disciplines.

TUF technology can be inserted at various levels of the system. For instance, it has been inserted at the operating system level of some systems [5, 1]. In this case, the MDS team is inclined to initially pursue a middleware solution—where the TUF scheduler, as part of the middleware, would manipulate the priorities used by the native scheduler in the system to affect system behavior—as a proof of concept. Potentially, the TUF scheduler might also modify other execution parameters, such as the frequency at which a given task runs or the particular method used to satisfy an executable goal.

At a minimum, a TUF scheduler can manage the behavior of the runtime system, utilizing information that complements that used by MPE during planning. Since the low-level scheduler monitors the actual execution environment, the TUF scheduler can have fairly good estimates of the processing needs of ongoing mission tasks, the frequency at which periodic tasks are run, and the available pool of processor cycles at the moment. This knowledge, combined with information concerning executable goals that are beginning or ending at a specific timepoint, can enable the TUF scheduler to determine the feasibility of the workload presented at the present time.

Moreover, it can potentially adjust scheduling parameters (such as priorities and frequencies) in order to influence the work that is actually done—and possibly increase the number of tasks that can be executed or increase the overall utility represented by the tasks that execute. It might also be able to signal the occurrence of an overload to the MPE scheduler (or to a cognizant MPE agent) before a goal fail-

ure would otherwise be detected.

Beyond these capabilities, the extent to which a dynamic scheduler can provide additional benefit is at least partially a function of the extent to which it can anticipate future workload and available processing resources. In particular, the granularity at which work is released to the lower-level scheduler from the MPE scheduler is a significant factor in determining the degree to which a TUF scheduler can reliably anticipate the future and, therefore, make scheduling decisions that are likely to optimize processor usage into the future.

Current thought indicates that, typically, MPE might produce a new MDS schedule, which is a network of partially ordered executable goals connected to nodes representing timepoints, every few minutes. Significantly, though, the arrival of a schedule does not necessarily indicate when any individual executable goal will be ready to run. This is revealed over time by MPE, which announces the release of individual timepoints as their associated state constraints are satisfied.

Part of the investigation and engineering that are taking place focus on the extent that future timepoints can be anticipated and the amount of work that can be done to prepare for their release.

4. The Organizing Principles

This section describes a set of organizing principles that have been applied to the MDS design and implementation, at least partially as a result of the addition of TUF technology to the system.

The set of organizing principles, when stated, seem straightforward. Nonetheless, many commonly used software design and coding practices do not necessarily adhere to these principles.

The following sections describe each principle in turn, offering some examples and commentary for each where appropriate.

4.1. Principle 1: Do Not Disrupt the Fundamental MDS Architecture

The MDS architecture addresses a specific set of issues concerning the modularity, integrity, flexibility, intellectual managability, and maintainability of systems, and it arguably represents a good solution. Consequently, the high-level architecture adopted for MDS should remain undisturbed when incorporating dynamic scheduling into the system.

Similarly, the current MDS design has implemented the architecture using MDS components that communicate by means of connectors. Once again, no modifications made during the incorporation of dynamic scheduling should make a substantial change to these building blocks. However, that is not to say that they cannot be modified. For instance, new ports can be added to components to facilitate the gathering of scheduling parameters or to provide adequate instrumentation or notification facilities.

4.2. Principle 2: Manage Processor Cycles to Maximize Mission Utility

A primary goal for the TUF scheduler is to apply processing cycles in a manner that best utilizes the available cycles to satisfy high-level mission goals. Specifically, under overload, an advantageous subset of the offered workload should be scheduled; and, when sufficient excess resources exist, the scheduler should opportunistically apply them to accrue the greatest realizable utility.

An important implication of this principle is the requirement that the two levels of MDS scheduling (MPE planning and the runtime scheduling) be sufficiently coordinated.

Prior to the introduction of TUF scheduling, MPE formulated plans based on user-supplied goal priorities. Runtime execution of threads was performed by a priority scheduler, using thread priorities assigned by a designer at mission-configuration time (or potentially later). Effective coordination of the two scheduling levels can be accomplished in at least two ways. First, if the workload is always sufficiently low, then the goals might be satisfied regardless of the specific priority assignment to threads. Second, if the workload is high enough that the priority assignment matters, then the priorities must be chosen so that an acceptable subset of the threads—or an acceptable order of all of the threads—is executed. This obviously requires the coordination of the priority assignments with a number of factors, including the user-supplied goal priorities and latency and schedulability requirements.

MPE planning remains unchanged with the introduction of the TUF scheduler. However, the benefits of the TUF scheduler—both in terms of coherently shedding work under overload, as well as opportunistically recognizing short-term scheduling decisions that can effectively utilize "excess" processor cycles—can best be served if the TUF scheduler can associate goal priorities with specific runtime threads. Since the TUF scheduler attempts to maximize accrued utility based on the TUFs provided to it, if the threads' TUFs are a function of the user-supplied goal priority, the TUF scheduler can be configured to maximize utility based on the workload presented to it by MPE.

Notice that the initial TUF scheduler, although it must ultimately associate an OS priority with each thread, might not use the goal priority directly as that priority for a number of reasons[1]. However, goal priority is a critical parameter

[1]For example, in a system where the workload is predominantly periodic, a technique such as rate monotonic analysis (RMA) might be used to

that can be taken into account when performing the thread-to-priority mapping so that the system can behave approximately as if it featured a native TUF scheduler.

4.3. Principle 3: Ensure That the System Will Continue to Function During Overloads

Historically, many space missions have fielded software that, despite the best efforts of designers and implementers, has defied attempts to bound the execution times for tasks. As a result, the software has been designed to accommodate occasional anomalous execution times by, say, incorporating a cycle-slipping scheme.

As missions become more autonomous in dynamic environments, it becomes even more difficult to anticipate the precise costs of actions (in terms of various resources), as well as the resources available, with certainty. For instance, a storm might deposit more dust on a rover's solar collectors than was anticipated, resulting in less available power for subsequent actions. Or the consistency of the surface might provide more or less traction than was anticipated, affecting the rate of power consumption.

Consequently, designers must anticipate that these systems will encounter processor overloads—times, possibly transient, when the demand for processor cycles exceeds the supply. During those times, the system must continue to function, albeit in a potentially degraded mode.

There are two distinct sets of functions to consider during operation under overload: the core resource management functions, which must continue to operate properly in order to manage the operation of the rest of the system, and the remaining functions.

The core resource management functions must be sufficiently decoupled from the remainder of the system that they will not be delayed unacceptably during overloads. For instance, the core functions should not directly execute arbitrary mission code, which might result in arbitrary delays. In addition, execution of the core functions must be governed by scheduling parameters that will ensure that they are executed in a timely manner, even under overload.

Mission applications must be written to cope with such uncertainty and possibly offer different processing options that can be employed as a function of load conditions. Many of these issues are dealt with in Section 4.4.

4.4. Principle 4: Be Prepared to Reduce Workload During Overloads

Given that processor overloads will occur, the system should be prepared to reduce the workload—at least on a temporary basis—to maintain control of the mission and to produce the best possible results under the circumstances. There are a number of approaches to reducing load, including simplifying tasks, reducing the number of tasks, dropping one or more iterations of a recurring task, and running some periodic tasks less frequently.

Before any of these steps are taken, the overload must be noticed. This can be done by a scheduler that can estimate and subsequently monitor the processor load; or the mission tasks can recognize overload conditions when their execution timelines are not satisfactory.

Once the overload is detected, it must be managed—a job that, for MDS, can be accomplished jointly by the MPE planner and the runtime scheduler. MPE can replan based on the current conditions, which will, itself, require some amount of time. The runtime scheduler must manage the system in the interim. For some common system structures, a priority scheduler might be used and the mission applications will switch modes to reduce processor load. Often such mode changes are quite dramatic; they are course-grained and result in a significant drop in workload—and in mission results. By contrast, TUF scheduling can potentially manage a much finer-grained degradation than other alternatives.

During normal operation in MDS, a TUF scheduler can make feasibility checks on schedules and attempt to shed executable goals intelligently as individual timepoints are reached, rather than responding once an overload has resulted in goal failures.

An even more capable TUF scheduler could modify scheduling parameters for executing threads. For instance, the priorities of specific runtime threads, while currently hardcoded in an MDS system, might be changed by a TUF scheduler.

Mechanically allowing such changes—that is, providing interfaces that would allow the TUF scheduler or an application manager to manipulate these scheduling parameters—is quite straightforward. However, ensuring that the system can continue to function properly while exercising such interfaces promises to be more difficult, since application code constrains what is possible in this respect. For instance, some components might accommodate a change in execution frequency, while others might not. The thread priority range that a given thread can execute under correctly might be implicitly or explicitly constrained on a thread-by-thread or component-by-component basis.

Consider also that MDS provides a rate group abstraction that allows the execution of a number of achievers—in a particular order—by a single thread running at a specified frequency. Rate groups exist, in part, because of the efficiency that they can provide by binding a number of computations with a single thread (thus conserving limited com-

assign the runtime priorities of a feasible workload. Or the priority space of the native operating system might not coincide with the priority space used for goal priorities.

puting resources). However, the desire for flexibility, which might benefit from the ability to alter the rate of specific achievers independently, argues for less, rather than more, aggregation into rate groups.

Taking advantage of dynamic changes to some of these parameters and mappings—to the extent permitted by the components and connectors that are in place—requires additional information from the application. This might include, for example, low-level utility information that related parameters such as execution frequency to quality-of-service parameters for a given achiever. Or mechanisms that allowed an executable goal to indicate to the TUF scheduler a range of acceptable qualities of service. Or, since cycle slipping and frequency adjustments both seem to be acceptable adaptations in general, interfaces could direct the scheduler to know which approach to favor overall for any specific task or task set.

Interfaces to address these and related issues enable the specification and construction of a more useful TUF scheduler—one that can dynamically modify scheduling parameters (within bounds) to allow more graceful behavior under high loads, perhaps avoiding goal failures while achieving additional application tasks.

4.5. Principle 5: Base Scheduling Decisions on Application-Provided Utility Metrics and Resource Consumption

MDS already features the foundation for making such scheduling decisions. The hooks that allow a TUF scheduler to be notified (a) when the MPE scheduler creates a new schedule—this is known as promoting the schedule—and (b) when a new timepoint in the current MPE schedule has been reached are in place in the MDS code.

Upon receipt of a schedule promotion event, logic in the proto-TUF scheduler traverses the MPE-generated goal network in order to determine the set of timepoints in the network. For each executable goal released at one of those timepoints, the scheduler determines the set of "top-level goals" to which it contributes, as well as the mission-assigned goal priority for each of those top-level goals. Furthermore, other logic identifies the state variable that is the subject of each individual executable goal and the constraints that are to govern that state variable.

This information provides much of the basis on which load shedding decisions can be made since it maps low-level executable goals to top-level goal priorities, which can be used when deciding which executable goals are most important to run at any given time. Moreover, in cases where two or more top-level goals might have a common MPE-assigned goal priority, the TUF scheduler could attempt to make sure that all of at least some top-level goals were accomplished, rather than part of many top-level goals, but none of any.

The rest of the information that a TUF scheduler might benefit from in the short-term involves the mapping from individual executable goals to the set of achievers that will act to satisfy each executable goal. The threads that will execute the achievers can then be identified and their resource requirements determined to a first approximation.

It is believed that this information can eventually be gathered mechanically—that is, under program control at compile-time, program-load-time, or at runtime. However, that is not yet the case. It must be done by hand now; and that will most likely continue to be the case for the duration of our project.

4.6. Principle 6: Reflect Available Concurrency as Separate Schedulable Entities

Mission applications are composed of tasks that must be executed. In MDS, these tasks are primarily the achiever logic that senses and affects the system's environment. Each task is bound to a schedulable entity, such as a thread, for execution. The runtime scheduler determines the order in which these threads are executed. If workload reduction is accomplished by preventing certain threads from executing—as might happen to the low-priority threads in a priority-based system—the unit of load shedding is the thread, and the amount of work shed depends on the tasks that are bound to it.

Providing a separate thread for each concurrent task in the system would promise to provide the potential to do the finest-grained load shedding. However, threads consume computer resources (e.g., stacks and OS control blocks), which are scarce resources in systems like MDS. Therefore, a tradeoff must be made that embodies a choice between having the finest granularity load management and conserving precious resources, which will typically result in some grouping of tasks to threads.

In a system that will shed load by preventing (or delaying) the execution of individual threads, it is clear that, when a set of tasks are mapped onto a single thread, the execution (or shedding) of all of those tasks should represent a consistent unit. As an extreme example, the single criterion used to determine the task-to-thread mapping, and the thread-to-priority mapping, would be the order in which designers would shed specific tasks.

Of course, that is not the only possible task-to-thread mapping strategy. For instance, in many predominantly periodic systems, all of the tasks that share a specific execution frequency are bound to a single thread, forming a rate group. In other cases, a server might handle requests from clients by means of a number of server threads; the number of threads indicates the maximum amount of true concurrency that the server could provide; but the actual tasks that

each thread performs depends on the nature of the client invocations that it services.

The situation might be even more complicated. For instance, in a system where there are computational stages—each featuring its own threads—it might make sense to shed groups of threads together. For instance, if the scheduler has determined that there is an overload and that the thread executing an actuator task will not be run, it might make sense to shed the tasks that gather sensor data in support of that task, assuming no other tasks need those sensor data.

MDS offers an additional complication. Since state variables *and their histories* are assumed to be available, it might make sense to execute sensor tasks that otherwise might be shed—just in case a later task might want to utilize the corresponding state variable's history.

These considerations merely illustrate that designers must take several issues into account when binding tasks to threads, including the implications of such binding on the granularity of load shedding.

4.7. Principle 7: Account for All Processor Cycle Consumption

A number of tasks consume processor cycles in computer systems. In real-time systems, it is important to account for all of them if dynamic schedulers such as TUF are to have accurate information on which to base their scheduling decisions.

For example, in the case of MDS, assume that the processor cycle requirements are known for all of the mission application tasks. There are other tasks in the system, such as the MPE planner, which run when needed and can consume significant processor resources when they do.

Similarly, the low-level scheduler consumes processor cycles, as do operating system services and daemons. For example, gathering the scheduling parameters needed by a TUF scheduler once an MPE plan has been promoted or a new timepoint has been fired requires processor cycles. Once again, there is a tradeoff here, where up-front processing by the TUF scheduler can be reduced by waiting until timepoints fire—introducing latency into the scheduler's decision in return for lower processor-cycle consumption.

4.8. Principle 8: Perform Adequate Analysis and Testing to Achieve Predictability

While dynamic scheduling techniques offer the promise of allowing greater flexibility, increased mission value, and more graceful degradation, they raise some difficult issues. By their very nature, they are dynamic and they are intended to operate in dynamic, somewhat unknown environments. Understanding their effects on the system in advance, so that the overall system can be characterized well enough to make its behavior predictable, is critical.

There are a few different aspects to this. First, once TUFs are created and assigned to tasks, system behavior under various conditions can be analyzed (formally or through simulations). Second, the characteristics of the scheduler itself must be understood—both its own execution (in terms of resource utilization and responsiveness), as well as its relationship to the MPE planner.

This particular area is the subject of an ongoing, three-year, internal MITRE research project that should produce formal results, methodology, and proof-of-concept software tools that can be applied to MDS.

5. Current Status and Future Work

The insertion of a TUF scheduler into MDS is still underway. Nonetheless, the preparation for TUF scheduling that has already taken place has resulted in a number of changes to the MDS code. While leaving the architecture and software architecture substantially unchanged, the low-level implementation of the MDS scheduling framework and the threading model used in the infrastructure have been modified to better accommodate dynamic scheduling, particularly under overload, based on the principles described in Section 4.

For the most part, these changes are not tied to the use of a TUF scheduler in particular. Rather, they are motivated by the desire to structure the system so that the scheduler is capable of executing when needed, regardless of the state of the mission applications and workload, and to manage processor cycles while considering mission priorities and actual resource consumption.

Future work will assess the ability of a TUF scheduler to provide near-optimal processor management in MDS' dynamic, remote environment.

6. Acknowledgments

A number of other MDS project members have been very generous with their time and insights, providing valuable understanding to the authors. We would particularly like to thank Kenny Meyer, Richard Morris, Robert Rasmussen, and Kirk Reinholtz.

References

[1] Mk7.3a release notes. The Open Group Research Institute Release Notes, 1998.

[2] M. B. Bennett and R. D. Rasmussen. Modeling relationships using graph state variables. In *Aerospace Conference Proceedings*, volume 5, pages 5–2085 – 5–2098. IEEE, March 2002.

[3] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley. An adaptive, distributed airborne tracking system ("process the right tracks at the right time"). In *Proceedings of the Seventh International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 1999)*, April 1999.

[4] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, School of Computer Science, Carnegie-Mellon University, August 1990.

[5] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146. USENIX, 1992.

[6] M. G. Gouda, Y.-W. Han, E. D. Jensen, W. D. Johnson, and R. Y. Kain. Distributed data processing technology, vol. iv, applications of ddp technology to bmd: Architectures and algorithms – chapter 3, radar scheduling: Section 1, the scheduling problem. Technical Report NTIS ADA047475, Honeywell Systems and Research Center, Minneapolis, MN, September 1977.

[7] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, May 1986.

[8] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J. Keleher. An example real-time command, control and battle management application for alpha. Archons Project TR #88121, Department of Computer Science, Carnegie-Mellon University, December 1988.

[9] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press, 1987.

[10] P. Regan and S. Hamilton. Nasa's mission reliable. *Computer*, 37(1):59–68, January 2004.