Resource Rationalizer: A Pattern Language for Multi-Scale Scheduling *

Christopher Gill and Douglas Niehaus †
and Venkita Subramonian
{cdgill,niehaus,venkita}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis

Lisa DiPippo and Victor Fay Wolfe
{dipippo,wolfe}@cs.uri.edu
Department of Computer Science
University of Rhode Island

Abstract

Assuring end-to-end timeliness properties in a distributed real-time system poses a variety of challenges at each, of several, levels of abstraction. The challenges vary with levels of abstraction that correspond generally to levels of architectural scale. In particular, there are key differences in the design forces for resource scheduling (1) within the operating system kernel, (2) within the middleware infrastructure on a single endsystem, and (3) distributed across endsystems in the entire system. While there is significant commonality related to the system's end-to-end timeliness requirements, the forces and their resolution vary non-trivially with each level of abstraction. Some design forces are seen at, or even span, multiple levels of scale. We document patterns that offer an organizing approach to design, within which the design forces at and across each level of abstraction can be resolved. This paper describes a pattern language for rationalizing resource scheduling at multiple levels of scale in distributed real-time and embedded systems.

Keywords: Real-Time Middleware and Operating Systems, Quality of Service Issues, Adaptive Resource Management, Distributed Systems.

1 Introduction

Many systems design paradigms, e.g., RMS [1], Low-Level Middleware Frameworks [2, 3], and POSIX [4], concentrate their attention at a single level of system abstraction. To achieve portability and generality, they often assume immutable properties at some other level of abstraction. However, fixing the mode of interaction of one level with an adjacent one inevitably limits the extent of cross-level integration available. This limitation reduces the designer's options to resolve design forces that cross level boundaries. This may result in inefficiencies and unnecessarily constrain the designer's degrees of freedom. For example, the POSIX priority-based thread management model is often cited [5] as functionally complete in that it is possible to implement any of the well-known scheduling paradigms by manipulating thread priority levels. However, the feasibility of such implementations can be significantly constrained by the time scales within which the priority manipulations will take effect.

This paper presents a pattern language, illustrated in Figure 1, for increasing coordination of resource management across multiple levels of architectural scale. In particular, it guides the designer toward greater degrees of freedom to achieve necessary end-to-end timeliness assurances for distributed real-time and embedded (DRE) systems. We examine patterns within, and bridging, each of three levels of abstraction for resource management within the context of DRE systems: (1) the operating system, (2) low-level middleware on a local endsystem, and (3) higher-level distributed middleware services that span endsystems.

OS Level: The operating system kernel has direct access to resources such as the CPU, network interfaces, and storage devices, and can perform fine-grain coordination of those resources, e.g., through scheduling interrupt handling, to achieve rigorous local timeliness assurances. The OS kernel also provides resource management abstractions such as thread scheduling models, e.g., the KURT-Linux Real-Time Scheduling Server (RTSS) or the POSIX priority-driven preemptive thread scheduling model [5], to software outside the kernel. These abstractions may be used both to obtain and restrict access to the

*This work was funded in part by The Boeing Company, the DARPA Quorum program, and ONR.

†Dr. Niehaus contributed to this work while on sabbatical from the University of Kansas.

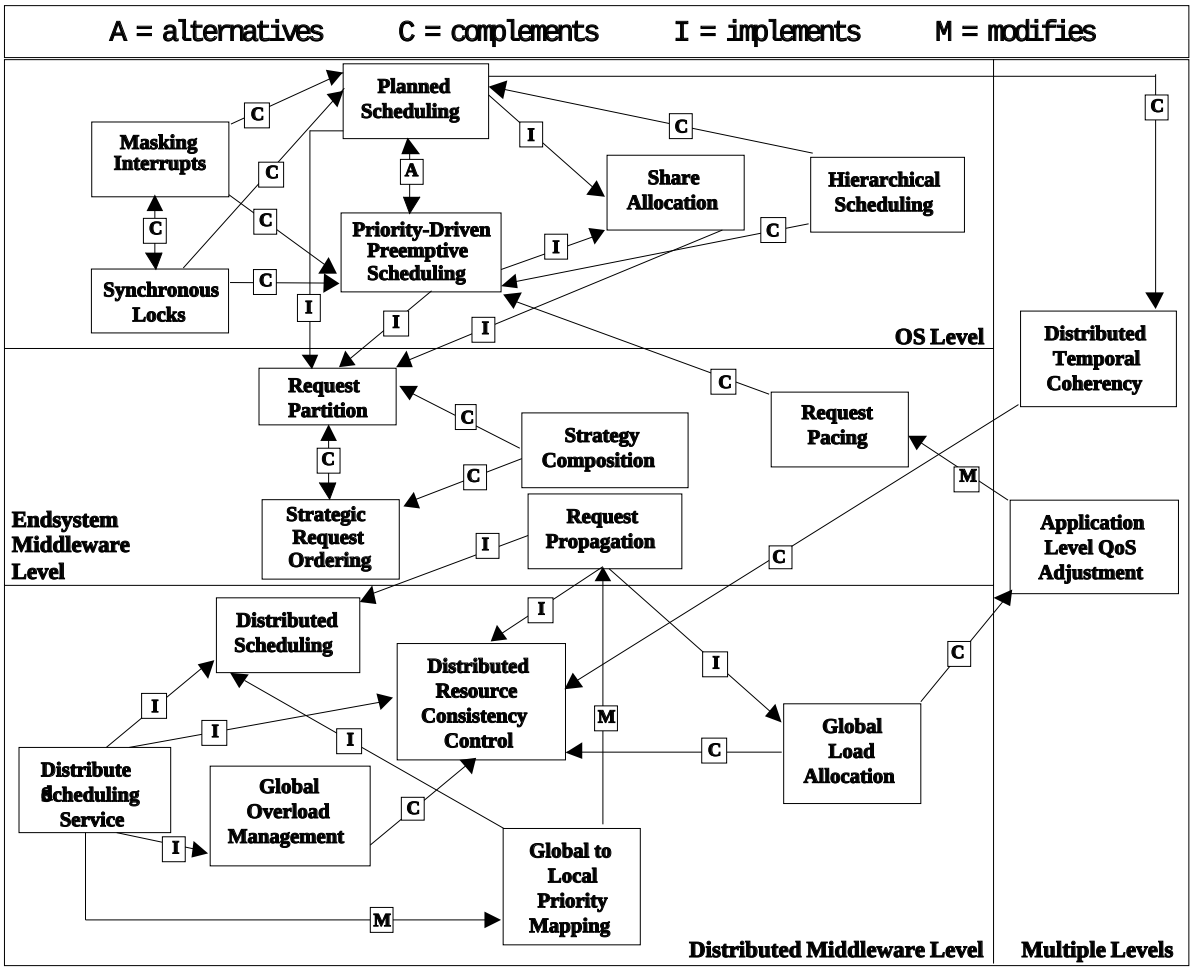

Figure 1: Map of the Resource Rationalizer Pattern Language

kernel-level system resources.

Endsystem Middleware: Above the operating system, low-level endsystem middleware frameworks such as ACE [2] and Kokyu [6] use abstractions from the operating system and other low-level middleware frameworks, to provide portable and consistent resource access across endsystem architectures. These low-level middleware frameworks provide abstractions for resource coordination within the local endsystem, which may in turn be used by higher-level middleware services.

Distributed Middleware: Higher-level middleware such BBN's Quo [7] and the URI Global Scheduling Service framework [8] must coordinate resource access across endsystems, to provide timeliness assurances, load balancing, load shedding, and admission control end-to-end in DRE systems.

To achieve rigorous end-to-end timeliness assurances for DRE systems, resource management must be coordinated both within and across each of these levels of abstraction. In applying the patterns described in this paper, the system designer is given latitude to balance the consequences of each design choice at each level. Returning to the previous POSIX example, using priority manipulation to implement the Request Partition pattern (Section 4.3) at the middleware endsystem level might be replaced at the middleware level, for example, by the Pacing Requests pattern (Section 4.1). Alternatively, in the context of an open source operating system, the designer could apply a different OS level pattern. For example, a system might use Planned Scheduling (Section 3.1) or Share Allocation (Section 3.3) instead of Priority-Driven Scheduling (Sec-

tion 3.2).

In this paper we examine design forces and patterns at each level of abstraction, and consider how additional cross-level patterns can be combined to form a pattern language we call Resource Rationalizer, for rationalizing end-to-end resource scheduling in distributed real-time systems. Figure 1 shows the interactions between patterns in the Resource Rationalizer pattern language. Most of the patterns are fundamentally about scheduling, though a few of the patterns (i.e., Masking Interrupts, Distributed Temporal Coherency) are needed for closure of the language, i.e., to resolve remaining forces not addressed by the other patterns. The additional patterns allow the language to produce generative designs for rational resource allocation, end-to-end in a distributed system.

The remainder of the paper is organized as follows. Section 2 describes the key design forces at each level of abstraction. Section 3 describes scheduling patterns at the kernel level. Section 4 describes scheduling patterns at the endsystem level. Section 5 describes scheduling patterns at the distributed scheduling level. Section 6 describes patterns applied to bridge between levels of abstraction and architectural scale. Finally, Section 7 draws conclusions about distributed real-time scheduling and the implications of this pattern language for developing distributed real-time and embedded systems.

2 Design Forces

The design forces addressed by this pattern language are summarized in Table 1, in the order in which they are discussed in this section. In describing a pattern language for multi-scale resource scheduling, careful analysis of the design forces the pattern language must resolve is crucial. Specifically, it is necessary to distinguish design forces that are part of the fundamental problem context from those introduced by particular design decisions made in the process of reconciling the overall system of design forces.

For example, rate monotonic analysis and assignment of task priorities [9, 10] is a mechanism commonly used to partition resource access requests into ordered groups, to ensure lower-frequency requests do not interfere with servicing higher-frequency requests. In certain cases, such as mutually exclusive use of resources shared among threads of different priorities, the semantics of priority-based resource allocation must be modified to ensure timeliness properties are maintained. For example, priority inversions [11] must be bounded to ensure resource allocation assurances are maintained so that stated timeliness assurances are not violated.

Table 1: Summary of Scheduling Design Forces

#	Force	Level(s)
F1	Temporally constrained request	All
F2	Request asynchrony	All
F3	Concurrency	All
F4	Performance constraints	All
F5	Space constraints	All
F6	Resource allocation semantics	All
F7	Constrained resource supply	Kernel
F8	Concurrent access to a resource	Kernel
F9	Request cost	Kernel
F10	Allocation granularity trade-offs	Kernel
F11	Resource utilization trade-offs	Endsystem
F12	Safety vs. interference	Endsystem
F13	Coordination and communication	Endsystem
F14	Encapsulation limitations	Endsystem
F15	Priority management	Endsystem
F16	Dynamic ordering	Endsystem
F17	Activities spanning endsystems	Distributed
F18	Multiple suitable resources spanning endsystems	Distributed
F19	Heterogeneity among operating systems' and endsystems local scheduling	Distributed
F20	Dynamic heterogeneous applications within an endsystem	Distributed
F21	Competing quality of service requirements	Distributed
F22	Abstract state consistency	Distributed
F23	Temporal consistency	Distributed

The key insight offered by this example is that the priority inversion problem *emerges* from the combination of: (1) the fundamental design force that some rational allocation of resources is necessary to ensure satisfaction of the specified timeliness constraints, (2) the design choice to use priority-driven preemptive thread scheduling, and (3) mutual exclusion semantics among threads due to the need for exclusive access to shared data.

Throughout this section we will examine fundamental

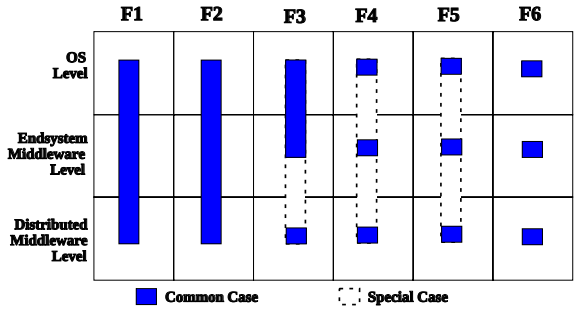

Figure 2: Multi-Level Scheduling Design Forces

design forces at the lowest level of system abstraction, the OS kernel, and design forces that emerge at higher levels as a result of the abstractions exposed at the lower level(s). Section 2.1 describes forces fundamental enough that they appear at multiple levels, and can across levels of abstraction. Section 2.2 examines resource scheduling forces at the OS kernel level. Section 2.3 describes design forces that emerge at the endsystem level. Finally, Section 2.4 considers design forces at the distributed services level.

2.1 Cross-Level and Multi-Level Forces

Some design forces are sufficiently fundamental that they appear at more than one level of the system. We present these fundamental forces first, to establish the overall context in which the Resource Rationalizer pattern language is applied. In some cases the force can be resolved entirely within a single level, using methods suited to the level in question. In other cases, the force manifests itself in a way that crosses levels of abstraction, and must be resolved by actions taken at multiple system levels. Figure 2 illustrates the forces that appear at multiple levels of scale. The usual span of a force is shown as a solid block in Figure 2, and special cases where the span may be broader are illustrated by dashed lines.

F1: Temporally constrained request. In addition to its cost, a request may bear a requirement for completion within a given interval, which may be absolute or relative in time. The time until a request completes may be a function of both its computation cost and of blocking or share reduction factors due to allocation of resources to other requests. The temporally constrained request force

F2: Request asynchrony. Not only can multiple requests arrive concurrently, in general the arrival pattern of requests may be inherently asynchronous. For example, at the kernel level these could be due to interrupts upon arrival of packets from a network or from other externally initiated events. At the endsystem level these could be due to asynchronous notification of a thread blocked on a condition variable. At the distributed level these could be due to method invocations from a remote client. This force not only appears at multiple levels, but it may also span all three levels, as in the asynchronous arrival of a method invocation request handled at the OS level as network packets, at the endsystem level as a queued upcall command in an ORB, and at the distributed level as a CORBA servant request.

F3: Concurrency. This design force is, perhaps, among the most fundamental, and has the largest influence on system implementation and behavior. Concurrency is desirable for several reasons, including ease of implementation and increased performance. For example, to achieve better timeliness of request completion and better utilization of resources, particularly with applications whose use of resources is reasonably complex in time, resources must be allocated to support, at least logically and possibly physically, concurrently executing application tasks. However, the benfits of concurrency come at the cost of requiring additional control in many cases, particularly those where concurrently executing threads share data whose semantics require mutually exclusive access. The need to ensure semantically correct use of shared data while also satisfying the performance constraints of every computation has an enourmous influence on system design and implementation.

F4: Performance constraints. To meet the timeliness constraints on a request, mechanisms and policies at all levels of abstraction must themselves allow adaptive reallocation of service requests to endsystems, or of lower level resources to requests within an endsystem, time consumption at the distributed services level must comply with the timeliness constraints of the application. Therefore, the load balancing, admission control, load reduction, and load shedding algorithms used by the distributed services must themselves operate within well-defined time constraints.

F5: Space constraints. In addition to limiting time complexity of distributed services, the amount of space

consumed, particularly in embedded systems, can be a limiting factor and dictate adoption of approaches that would otherwise not be chosen. For example, assigning and managing global priorities for end-to-end timeliness constraints can reduce the amount of state information maintained on each endsystem, compared to using planned schedules, thus favoring priority schemes in situations where the space consumed by the execution plan is significant.

F6: Resource allocation semantics. In addition to limiting time complexity of distributed services, the range of ways in which a resource can be allocated while maintaining required semantics is important.

2.2 Kernel Scheduling Forces

We start at the lowest level of resource abstraction, within the operating system kernel. We defer discussion of additional kernel-level design forces induced by design choices at the OS kernel level until Section 3. The fundamental design forces for resource management at this level of scale are as follows.

F7: Constrained resource supply. Inherent in the definition of a shared resource is that access to it is constrained in time, and possibly in total quantity. For example, concurrent access to a CPU is constrained in time, while energy consumption may be constrained both in time (power) and quantity (battery charge).

F8: Concurrent access to a resource. Multiple requests contend concurrently for the resource, so that depending on resource access granularity, requests must either take turns accessing the entire resource, or receive partial shares of the resource over a given interval.

F9: Request cost. To service a request, a certain share of the resource must be granted to the request over a given period of time. In some cases the total cost of a request is insensitive to the pattern of resource allocations over time. In others, e.g., voltage scheduling of power-aware CPUs [12], the cost is a more complex function of the actual resource allocation over time.

F10: Allocation granularity trade-offs. Finer granularity allocation of a resource increases the level of control the system has to produce timely results, but comes at the price of greater overhead, which reduces the utilization of the resource for productive work. In the case of the CPU resource, for example, non-preemptive thread scheduling increases the activation latency for a newly *ready* thread until at least the next yield point in the currently executing thread. In preemptive thread scheduling, the operating system is free to switch to the newly ready thread immediately, but the frequency with which the system chooses to switch among threads determines the portion of the CPU resource used for the unproductive work of switching context among threads.

2.3 Endsystem Scheduling Forces

The next level of abstraction for resource scheduling resides above the operating system kernel but within a single endsystem. We describe both design forces inherent to the domain of endsystem scheduling, and design forces induced by choices at the OS kernel level. Some of the fundamental design forces for resource management at the endsystem level are as follows.

F11: Resource utilization trade-offs. As the endsystem scheduling infrastructure is architecturally closer to the application than the OS, it is reasonable to place more awareness of the application itself, i.e., at least the specific projection of the application into one endsystem, into the low-level endsystem middleware. Therefore, the endsystem scheduling infrastructure will necessarily be responsible for managing the inherent trade-offs between increasing the amount of useful work performed overall, and achieving necessary timeliness assurances for completion of individual application tasks. However, the ability of the endsystem to do this may be significantly constrained by the scheudling capabilities exposed to the endsystem level by the OS level.

F12: Safety vs. interference. With the addition of concurrency, access to resources must be ensured to be safe with respect to timeliness requirements, due to possible interference between tasks. Because the operating system is in general unaware of the structure of resource requests made by the application, it is very difficult to control resource protection efficiently within the operating system without at least some hints about that structure from the endsystem level scheduling infrastructure.

F13: Coordination and communication. While strict isolation of resources among concurrent tasks may fa-

cilitate safety of concurrent resource use, it may in turn hamper computation overall, as concurrent tasks are restricted in sharing results. In the endsystem scheduling domain, coordination of resource requests and exchange of computation results among concurrent tasks must be managed.

In addition to the above inherent design forces, additional design forces may be introduced by the design choices made at the operating system level. Additional forces relevant to the endsystem scheduling domain include the following.

F14: Encapsulation limitations. Some choices at the OS level propagate unavoidable constraints to higher levels of abstraction, as noted in Section 2.1. Each particular resolution of design forces at the OS level of abstraction may induce different semantics for safety and performance at the endsystem middleware level. For example, in the case of non-preemptive thread scheduling, blocking latency for access to the CPU must be considered whether or not synchronization abstractions, e.g., thread mutexes, are used by the middleware or the application.

F15: Priority management. If the operating system exposes a primarily priority-based interface for arbitration of thread access to the CPU, as is the case in most commercial-off-the-shelf (COTS) operating systems and as specified by the POSIX standard [4], then priority management may be the only reasonable mechanism for meeting static end-to-end timeliness requirements. Furthermore, priority-based management may also be useful to constrain the time or space complexity of distributed scheduling, with priority-based endsystem scheduling being a natural basis for implementing the end-to-end priority approach. However, using priorities to allocate resources may result in a new set of issues related to a semantic "impedance mismatch" between the priority mechanism and the fundamental application semantics. For example, priority-based schedulers do not explicitly consider time, so blocking times and other common factors may increase the complexity of analysis, and possibly reduce the achievable degrees of assurance, for meeting crucial timeliness requirements. Instead, explicitly specifying an execution schedule to meet some kinds of timeliness requirements may be simpler than orchestrating a priority assignment scheme to achieve the same assurances.

F16: Dynamic ordering. An alternative to static resource management is to select among resource requests dynamically. Furthermore, in some cases, e.g., when request deadlines are not specified until the moment of arrival, static resource allocation is simply not possible.

2.4 Distributed Scheduling Forces

The distributed scheduling forces described here will act upon systems in which uniformity across the system is essential to ensure predictability. These forces may act upon local endsystems, the entire system globally, and on possibly all points on the continuum between these two endpoints.

F17: Activities spanning endsystems. Each endsystem represents the outer limit of resource allocation scope achievable at lower levels of architectural scale. However, activities such as chains of remote method invocations may traverse many such scopes, and require coordination of scope-by-scope resource allocation assurances to achieve overall enforcement and analysis of real-time requirements.

F18: Multiple suitable resources spanning endsystems. In applications that span endsystems there may be choices as to which endsystem to assign a task, or which resources within various endsystems to allocate to the task. Distributed scheduling should allocate resources from appropriate endsystems to application tasks in a way that facilitates overall enforcement and analysis of real-time requirements.

F19: Heterogeneity among operating systems' and endsystems' local scheduling. Multiple OS's and/or endsystems, each with importantly different scheduling policies, may be involved in an application that spans these subsystems. When referring to something that can apply to multiple operating systems and/or endsystems, we will use the term "subsystem". If each subsystem provides multiple scheduling policies and/or scheduling parameters, it is possible that the global effect of the local scheduling choices may be undesirable due to conflicting forces in the enclosing design context. For example, if deadline-based and priority-based scheduling were used in two different subsystems, it may be difficult to rationalize scheduling requirements in applications that span them [13].

F20: Dynamic heterogeneous applications within an endsystem. Within endsystems where tasks implementing heterogeneous computations may dynamically enter the system or change their requirements, it may be necessary to abstract scheduling policies from the endsystem to a global service that is used by all entities, to ensure a compatible notion of policies and parameters among the heterogeneous application tasks.

F21: Competing quality of service requirements. Applications may specify a variety of Quality of Service (QoS) requirements, each of which should affect scheduling. For instance, timeliness requirements such as deadlines and periods can affect the order of execution. However, many systems consider other parameters that can affect computation scheduling. *Criticality* is often used as a parameter in addition to priority to help represent relative importance of computations to help resolve allocation conflicts, particularly in overload situations.Some systems use a basic mandatory/optional Criticality specification, others use a ordinal Criticality value per task, still others use utility functions to specify Criticality.

Other QoS requirements such as Security, Accuracy, and Fault Tolerance all may also impact the scheduling. Furthermore, these QoS requirements may conflict. For instance, achieving timeliness may require sacrificing accuracy, or vise versa. Similarly, tradeoffs arise among most QoS requirement categories:timeliness and security, security and accuracy, timeliness and reliability. Distributed scheduling decisions should consider global QoS requirements and tradeoffs.

F22: Abstract state consistency. While it is not possible to maintain a completely accurate, up-to-date picture of the global state of an entire distributed system, it is necessary to maintain a view that is *consistent*, *i.e.*fundamental properties such as causality are not violated, within some level of precision. This view may simplify the problem by only keeping abstract state information about key system properties, e.g., the remaining cost of a multi-endsystem transaction. Consistency of the abstract state is necessary in order to ensure schedulability across the distributed system.

F23: Temporal consistency. As a special case of maintaining consistency of abstract state, dynamic allocation of resource requests across multiple endsystems may require that temporal consistency, through the use of clock synchronization or of virtual clocks [14], be maintained among those endsystems to assure timeliness properties end-to-end.

3 Kernel Scheduling Patterns

Scheduling patterns at the OS kernel level of abstraction deal with the inherent design forces raised by limitations on the use of resources and on the resource semantics, combined with design forces raised by semantics at higher levels of abstraction. We document these patterns according to Coplien's [1] format, using the name of each pattern as its subsection title. This section is organized as follows: Section 3.1 presents the Planned Scheduling pattern, which in open kernel implementations offers a natural semantics for specifying resource allocation policies to enforce timeliness requirements; Section 3.2 describes the Priority-Driven Scheduling pattern, which in COTS POSIX-based operating systems provides another semantics for specifying resource allocation policies; Section 3.3 documents the Share Allocation pattern, which allows subdivision of a resource among competing activities, and can be implemented using either the Planned Scheduling pattern or the Priority-Driven Scheduling pattern; Section 3.4 presents the Hierarchical Scheduling pattern, which can apply a series of different scheduling patterns to successively refined groups of tasks until a single task is selected to run. This gives different resource allocation semantics to different resource usage domains. Section 3.5 documents the Masking Interrupts pattern, which reconciles the desire to share critical sections across process and interrupt contexts, with the need to prevent unsafe interleaving of critical section invocations. Finally, Section 3.6 describes the Synchronous Lock pattern, which reconciles the desire to share critical sections among software executing concurrently on separate processors in a multiprocessor machine.

3.1 Planned Scheduling

Problem: Commonly available priority-based implementations may impose semantics that are insufficiently congruous with the application semantics. Consider the semantics of a control system in which operational data

[1] http://hillside.net/patterns/definition.html

arrives at a specific interval, and the control calculation uses the most recent data to modify set-points of specific devices. In a priority-driven system where the calculation is blocked on the arrival of the data event, jitter in the data arrival is reflected in subsequent execution jitter of the control law. In contrast, the explicit plan runs at a specified frequency, but is free to handle the data arrival jitter in a way that does not transfer it into execution jitter.

Context: Real-time systems in which computations need to receive particular qualities of service, using open operating system kernels.

Forces: The constrained resource supply, resource cost and competition for the resource forces combine with the temporally constrained request, request cost, and request asynchrony forces to require temporal arbitration of resource access among requests. In contrast to the Priority-Driven Scheduling pattern described in Section 3.2, earlier binding to the time-line is desirable, to enforce constraints such as equivalent isolation of independent computations.

Solution: Execution behaviors of computations in the system are naturally expressed as intervals of execution placed on the system timeline. Therefore, explicit planning of schedules that model the desired execution behaviors is often, where possible, desirable. These intervals may be defined using explicit times, as in classical time-driven scheduling approaches [15], or parameterized by relative times of other tasks [16]. Examples of the planned scheduling pattern appear in KURT-Linux [17], clock-driven schedulers, and Maruti-II [18].

Resulting Context: If planned schedules are used, the application or a system-wide scheduling service must describe computation behavior in explicit terms, so that the operating system can synthesize a complete schedule of execution, for the system as a whole, at the operating system level. The Hierarchical Scheduling pattern described in Section 3.4 may be applied to address jitter in request arrival that cannot be handled by a planned schedule, e.g., by applying the Priority-Driven Scheduling pattern as a secondary scheduling layer. Furthermore, the choice of execution intervals in a planned schedule specifies usage granularity of the resources it controls.

Rationale: The design makes explicit the order of execution of requests, simplifying validation of timing constraints. The pattern allows natural expression of a schedule meeting specified constraints, *in the same terminology as the constraints themselves.*

3.2 Priority-Driven Scheduling

Problem: A generic operating system implementer needs to provide a functionally complete interface for resource allocation, without adding noticeable programming model complexity.

Context: Real-time systems in which computations need to receive particular qualities of service, using COTS POSIX-based operating system kernels.

Forces: The constrained resource supply, resource cost and competition for the resource forces combine with the temporally constrained request, request cost, and request asynchrony forces to require partially ordered arbitration of resource access among requests. In contrast to the Planned Scheduling pattern described in Section 3.1, later binding to the time-line is desirable, to strike a balance between data arrival and execution jitter, quality of service, and isolation forces. In addition, the application programmer desires a familiar and easy-to-use interface. For system performance, or due to less reliable information, a quick and efficient decision function is also desired.

Solution: Implement priority-based scheduling at the OS level, to provide a mapping between the integers (simple and familiar interface) and priority-based thread execution semantics. Examples of this pattern appear in POSIX-compliant UNIXes [5], Windows NT [19], and OS/2 Warp [20].

Resulting Context: Analysis techniques such as RMA are needed to map fundamental task properties such as periodicity onto a priority assignment capable of enforcing application timeliness requirements. These analysis techniques may require adjustments depending on whether priorities are enforced preemptively or non-preemptively.

Rationale: The design simplifies the information used to express and enforce access to resources, often to an integer representation. The pattern provides a means to validate and enforce timing constraints indirectly, through analysis of the resulting schedule.

3.3 Share Allocation

Problem: Some constraints are best expressed as shares of a resource over an interval, and commonly available OS-level abstractions are for explicit plans or priorities rather than shares.

Context: Applications using either planned or priority-based scheduling.

Forces: The constrained resource supply, resource cost, and competition for the resource forces combine to require managed resource allocation to ensure resources are shared accurately. Furthermore, the allocation granularity force constrains the intervals over which share assurances can be made.

Solution: Using either priorities or explicit schedules, shares can be specified and enforced. Different applications may receive different shares. Furthermore, the fidelity of the actual allocation to each share requirement may also differ, due to the interaction of allocation granularity with the requested share interval. Examples of this pattern appear in fair queuing network routers and time-space partitioning architectures [21].

Resulting Context: Differences in the semantics of the priority and explicitly planned scheduling approaches propagates upward to the resulting share-based semantics. For example, for completely CPU-bound executions the semantic differences may not be noticeable as there are no differences in the un-utilized resource intervals. However, if there are intervals in which the CPU is unused, then the placement of those intervals may induce semantic conflicts in the subsequent periods. In planned scheduling, tasks that are not ready at the arrival of their planned execution interval are penalized, while in priority scheduling the lowest priority tasks may pay the penalty for any higher priority task delay.

Rationale: The design ensures isolation of an application's resource requests from those of other applications. The pattern also supports isolation of resource requests within an application, *e.g.*, between concurrent threads.

3.4 Hierarchical Scheduling

Problem: Appropriate resource allocation semantics may vary across hierarchical levels of execution semantics. For example, ensuring time-space isolation of processes may require explicit scheduling of CPU shares, while within a process individual threads may require concurrency control related to application semantics.

Context: Applications whose computation scheduling semantics may be decomposed into and coordinated across two or more hierarchic levels of resource allocation.

Forces: The competition for the resource force occurs at multiple hierarchical levels. Furthermore, the most natural semantics to describe resource allocation can be widely different at each level. Finally, the resulting context of a particular OS-level scheduling pattern may be immediately resolved by applying another complementary scheduling pattern.

Solution: Apply scheduling patterns hierarchically, with the lower level scheduler delegating execution to the next higher-level scheduler once its constraints are satisfied. Examples of this pattern appear in the Spring Kernel [22], KURT-Linux, and RED-Linux [23].

Resulting Context: A particular implementation of multi-level scheduling may introduce various additional forces at all levels of abstraction. For example, the order in which constraints are applied may have implications for endsystem-level scheduling [6] and for end-to-end scheduling.

Rationale: The design allows scheduling strategies to be applied selectively to different parts of a system. The pattern can serve to rationalize scheduling for distinct but interrelated portions of a system, *e.g.*, for network flow fairness or coordinated scheduling of groups of collaborating processes.

3.5 Masking Interrupts

Problem: Interleaved execution of system code and interrupt handler code can cause inconsistency of shared data. For example, device drivers commonly contain code accessing shared data within system calls executed in process context, and within device handlers executed in interrupt context.

Context: Critical section shared among code executed under scheduler control and in interrupt context.

Forces: The competition for the resource force combines with the resource allocation semantics, exclusive access to the shared data, and request asynchrony forces to produce possibly unsafe executions in which access to shared data by the asynchronously invoked computations must be controlled to maintain consistency constraints.

Solution: Mask interrupts temporarily to execute critical sections of code. For example, a common approach on COTS platforms such as those using the x86 processor family is to use the CLear Interrupts (CLI) and SeT Interrupts (STI) instructions to mask interrupts temporarily during critical section execution, in either system call or interrupt handler context.

Resulting Context: The CLI/STI approach is an effective form of concurrency control on single CPU system, but must be combined with the Synchronous Locks pattern described in Section 3.6 for systems with more than one CPU. Furthermore, coarse-grained interrupt control can significantly impact scheduling jitter. If there are only two interrupt levels, and any driver that needs to control concurrency by blocking interrupts interferes with the scheduling interrupts, significant levels of execution jitter can occur, potentially impacting timeliness constraints. With multiple interrupt levels, or careful narrowing of the critical sections, lower levels of jitter may be achieved. For example, on platforms such as 68K family CPUs, multiple interrupt levels exist, enabling the system to separate the scheduling interrupt at a higher level than those used by devices, thus isolating scheudling functions from interference by the device interrupts.

Rationale: The design allows control over asynchronous forms of concurrency, *i.e.*, due to hardware interrupts. The pattern masks interrupts temporarily, and ideally induces minimal scheduling jitter either through inherent separation of interrupt levels or careful narrowing of contention intervals.

3.6 Synchronous Locks

Problem: Concurrent executions of code accessing the same data can result in corrupt data. For example, concurrent execution of the task scheduler by more than one CPU can corrupt the task list or cause selection of the same task to execute concurrently on more than one CPU, unless preventive measures are taken. Similarly, in a multi-threaded server process, assignment of jobs to threads by a scheduling routine must protect the job queue to avoid corrupting the queue or having two threads attempt to serve the same job.

Context: Global or shared data accessed by multiple threads of control executed concurrently.

Forces: The competition for the resource force combines with the resource allocation semantics force (exclusive access to the shared data) and the concurrency force to produce possibly unsafe executions in which access to shared data by the physically concurrent computations must be controlled to maintain consistency constraints.

Solution: A common approach is to employ semaphores managed by locks acquired and released synchronously within each concurrent thread of execution (*synchronous locks*), to manage the shared data resource. The appropriate semantics of the lock may vary according to efficiency and correctness considerations such as physical concurrency, self-deadlock, and intervals of contention for the shared data.

We note that this pattern is strongly related to the Masking Interrupts pattern in Section 3.5, as both serve to control concurrency. The Masking Interrupts pattern controls concurrency arising from asynchronous arrival of interrupt signals transferring control to interrupt handler code, while the Synchronous Locks pattern is used to control physical or logical concurrency arising from multiple threads of execution. It is also important to note that *both* patterns apply in uniprocessor and multiprocessor operating systems, because both types of concurrency are present, but only the Synchronous Locks pattern applies in multi-threaded user code, because interrupt handlers execute only in the operating system context.

Resulting Context: A designer may need to consider additional details of the particular context within which the Synchronous Locks pattern is applied, to identify the most appropriate form of locking to employ in implementing the pattern. For example, multiple CPUs within a machine can concurrently execute both user and system code efficiently, by allowing physical concurrency of threads.

A spin-lock does not cause the calling thread to context switch of it does not obtain the lock, rather the CPU actively waits for the semaphore controlling the resource

to be freed. However, a spin-lock may *only* be used with physical concurrency, as it will exhibit self-deadlock in a single-CPU concurrency architecture. Furthermore, a spin-lock is desireable if and only if the resources in question are generally held for extremely short times, so that the cost of a context switch exceeds the cost of waiting. When a spin-lock is not appropriate, *e.g.*, due to the resource being held for longer intervals relative to the context switch interval, a blocking lock should be used. Depending on properties, such as fairness or prioritization, desired for the order of access to the lock by blocked threads, a blocking lock may also employ various queueing disciplines on waiting threads to enforce those properties.

Rationale: The design trades efficiency for correctness, balancing safety and liveness properties of concurrency [24]. The pattern allows several variations to improve efficiency while preserving correctness under different conditions of physical concurrency and relative lengths of contention and context switch intervals.

4 Endsystem Scheduling Patterns

Scheduling patterns at the endsystem middleware level of abstraction must deal with the design forces raised by both the OS level and the Distributed Services level. This section is organized as follows: Section 4.1 documents the Request Pacing pattern, which allows the endsystem scheduling infrastructure to influence the overall resource usage behavior; Section 4.2 presents the Request Propagation pattern, which addresses accounting for local resource usage within multi-endsystem requirements at the distributed services level; Section 4.3 describes the Request Partition pattern, which allows the endsystem to isolate the resource requirements of one group of requests from another, and achieve different qualities of service for each group; Section 4.4 documents the Strategic Request Reordering pattern, which allows tailored permutation of resource requests to improve quality of service, end-to-end and within the endsystem; Finally, Section 4.5 presents the Strategy Composition pattern, which allows combinations of other endsystem scheduling patterns to be composed coherently, in a similar way to the Hierarchical Scheduling pattern described in Section 3.4.

4.1 Request Pacing

Problem: When operating near the performance envelope of an operating system, overload and contention for resources may be exacerbated by particular resource usage behaviors above the operating system level of abstraction. This is a general phenomenon, but is more prevalent when real-time semantics were not a first-class design criterion of the supporting operating system.

Context: Applications with stringent timeliness requirements hosted on COTS operating systems.

Forces: The resource utilization trade-offs force combines with the encapsulation limitations force to constrain the rate at which requests can be made and still attain a particular quality of service. Spacing of resource requests in time is a major influence on utilization and availability of resources within an endsystem. Explicit control using the OS API to separate resource requests may be tedious and error-prone, particularly if the application semantics and underlying OS resource allocation semantics differ significantly. Instead, explicitly pacing requests at the endsystem middleware level induces separation of allocation requests at the OS level.

Solution: Pace requests for resources to separate them in time, reducing contention and improving resource allocation assurance without constantly manipulating the OS API. We thus modulate the resource requests coming to the OS level from higher levels of abstraction, to return the OS to a more stable operating mode.

Resulting Context: In general, pacing can be used to reduce latency jitter in servicing resource requests, and thus provide a more consistent allocation sequence overall. Some choices at the OS level, e.g., use of priority-driven preemptive thread scheduling may be impacted as well, with either greater or lesser context switching overhead, depending on the resulting arrival sequence for requests. Self-pacing may lead to drift in the timeline, whereas pacing with respect to an external time standard [25] can maintain greater consistency overall.

Rationale: The design emphasizes application-level changes in behavior to modulate overall system behavior. The pattern guides an application to behave as a “good citizen” in the larger context of a shared-resource system.

4.2 Request Propagation

Problem: End-to-end timeliness requirements cannot be satisfied entirely within a single endsystem.

Context: Distributed real-time applications in which activities span multiple endsystems.

Forces: The coordination and communication force combines with the temporally constrained request force and the resource allocation semantics force to require endsystem-by-endsystem accounting for end-to-end requirements. Resource allocation on an endsystem may modify the remaining requirements for the activity as it proceeds to subsequent endsystems. Furthermore, values of parameters used to enforce compliance with the modified requirements may need to be updated as well. For example, execution cost for a portion of a distributed transaction executed on one endsystem must be deducted from the remaining expected cost of the transaction on subsequent endsystems.

Solution: Parameterize end-to-end requirements with values that can be mapped to and measured on each endsystem. Update parameters on each endsystem as the activity progresses, thus tailoring the requirement to subsequent endsystems.

Resulting Context: Communication overhead and other sources of delay along the end-to-end path must be considered in updating the requirement. Work that reduces the cost on subsequent endsystem should be distinguished from overhead that does not contribute to progress. To construct a coherent piecewise mapping of parameters from endsystem to endsystem, parameter adaptation techniques may needed [13]. The Request Propagation pattern may be applied in implementing the Distributed Scheduling (Section 5.2), Distributed Resource Consistency Control (Section 5.3), or Global Load Allocation (Section 5.5) patterns. In addition, for priority-based endsystem scheduling, the Global to Local Priority Mapping pattern (Section 5.4) can be applied to configure the translation between local and global parameter values.

Rationale: The design emphasizes local accounting for end-to-end management of timeliness and resource access. The pattern offers a way to rationalize scheduling of requests propagating across multiple local endsystems.

4.3 Request Partition

Problem: Resource requests from one group of tasks may interfere with those of another, and without some form of mediation it is not possible to enforce a rational policy for separating the impact of one group of requests from the other.

Context: Applications with differing quality of service (QoS) requirements for different tasks on an endsystem, particularly where at least one group of tasks requires stringent QoS assurances that are at risk from interference by other groups of resource requests.

Forces: The safety vs. interference force combines with the concurrency force to require isolation of the effects of one task's resource requests on the servicing of another's requests. Tasks compete for shared resources that are constrained in their availability. Allocating a resource to one task may delay or otherwise interfere with the ability to allocate the resource to another task.

Solution: Provide policies and mechanisms for isolating resource requests made by one group of tasks, from the requests made by another group of tasks. For example, the Share Allocation pattern described in Section 3.3 can be used to prevent one group of tasks from exceeding its allocated ration of the resource, and ensure that another group will not be impacted by an excessive number, or clustering, of resource requests by that group.

Resulting Context: Priorities are often used on COTS POSIX-based endsystems to implement request partitioning, *e.g.*, Real-Time CORBA 1.0 [26] (RT CORBA) priority lanes in TAO [27]. This simplifies the decision function for allocation, and allows other optimizations such as eliminating the queuing overhead seen for the Request Reordering pattern described in Section 4.4, if it is sufficient to dispatch requests in order of arrival within each priority level. Planned scheduling offers another way to provide resource partitioning that may improve performance in some cases, e.g., if significant thread context switching was needed to enforce preemptive priority levels.

Rationale: The design emphasizes isolation of resource requirements of one set of requests from another by partitioning the requests into equivalence classes. The pattern allows different kinds of isolation relationships between

the equivalence classes, *e.g.*, priority-based vs. share-based, to provide different kinds of resource allocation assurances.

4.4 Strategic Request Reordering

Problem: While priorities or planned schedules may be used efficiently to isolate resource requests between task groups, arrival of requests for resources may differ from the expectations under which those mechanisms were applied.

Context: Applications in which resource requests arrive in a sub-optimal order, for which improved resource allocation capacity or timeliness assurance can be achieved by reordering the requests.

Forces: The temporally constrained request force combines with the dynamic ordering force to require that requests be reordered at key scheduling points in the system, e.g., where multiple network connections are multiplexed onto a single endsystem thread. Reordering requests may be expensive, both in fixed overhead and in increased time complexity. Reordering that can be performed with better worst-case overhead bounds improves real-time assurances that can be given in the face of reordering to meet application timeliness requirements. Depending on the resource request behavior of a particular application, different policies and mechanisms for reordering may give better best-case, average-case, and/or worst-case bounds on overhead.

Solution: Allow different policies and mechanisms for reordering resource requests on an endsystem. For example, if requests are ordered monotonically once enqueued, as for deadline aging in the Earliest Deadline First (EDF) scheduling strategy, a queue may be used efficiently to reorder requests. If a reordering strategy is based on a known and well bounded population of possible values for reordering decision function parameters, then hashing can be applied to reduce the overhead of reordering [28, 6].

Resulting Context: Reordering delays may impact overall resource allocation latency, and thus impact feasibility of timeliness assurances. Reordering delays may also introduce jitter in latency of servicing requests. When this pattern is applied, both of these factors must be considered in overall schedulability analysis and enforcement policies on an endsystem. Furthermore, there are practical limits on the time-scales for which request reordering is applicable.

Rationale: The design respects the possibility that requests may arrive at an endsystem incorrectly ordered for dispatching in a way that meets timeliness requirements. The pattern allows scheduling of streams of requests arriving out of order or from different sources, to be rationalized on the local endsystem.

4.5 Strategy Composition

Problem: A scheduling pattern that is preferable to address one resource access requirement may be poorly suited to address other requirements.

Context: Applications with multiple resource access requirements, each of which is best addressed by a different endsystem scheduling pattern.

Forces: The encapsulation limitations force combines with other forces, e.g., the priority management and dynamic ordering forces, to require a composite scheduling approach. Requirements may be ordered, with the constraints imposed by one taking precedence over those of another. Constraints that succeed others must be enforced in a manner that is stable and meaningful with respect to the enforcement of the preceding requirements.

Solution: Apply endsystem level and OS level scheduling patterns in an ordered manner, similar to the Hierarchical Scheduling pattern described in Section 3.4. Unlike the Hierarchical Scheduling pattern, the Strategy Composition pattern places less emphasis on the nesting of composed patterns, but rather focuses on the stability relationships among them. For example, the MUF [29] scheduling strategy may use the preemptive form of the Priority-Driven Scheduling pattern at the OS level to implement the Partition Requests pattern at the endsystem middleware level, and then use separate queues within each priority to implement the Reorder Requests pattern in a way that does not disturb the request isolation between priorities.

Resulting Context: Ensuring that the composition of strategies exhibits necessary stability properties may add complexity and time cost to the analysis of the resulting composite strategy. Strategies whose stability properties are invariant with respect to run-time factors are

best suited to adaptive scenarios where resource feasibility must be recomputed at run-time. However, new strategies may be synthesized to support new scheduling paradigms by applying this pattern, such as the use of RMS+MLF to support imprecise computations [30].

Rationale: The design leverages the semantics of individual scheduling strategies. The pattern allows individual strategies to be composed in a way that respects the semantics of individual strategies, but rationalizes the combined effect.

5 Distributed Scheduling Patterns

All of the patterns in this section address distributed real-time systems where the application has tasks with real-time constraints that span operating systems and/or endsystems (subsystems) that have possibly different scheduling policies, and the designers must be able to predict the real-time behavior of the entire system to some reasonably high level of assurance. An example of a real-time task spanning subsystems is a client application from one subsystem requesting service from a servant in another subsystem under a deadline. We assume that the local OS's and endsystems use the scheduling patterns from Sections 3 and 4 respectively, but that to facilitate enforcement and analysis of real-time requirements across the entire system, the service-level patterns of this section are needed to provide a uniform application of the choices of policies and parameters provided by the local OS's and endsystems.

This section is organized as follows. Section 5.1 presents a Distributed Scheduling Service Pattern, which may be applied, in implementing several of the other patterns: Distributed Scheduling (Section 5.2), Distributed Resource Consistency Control (Section 5.3), Global To Local Priority Mapping (Section 5.4), and Global Overload Management (Section 5.6). The Distributed Scheduling Service pattern is not the only way to provide the coordinated scheduling required by those other service-level patterns, but is the only one presented in this paper. The Global Load Allocation pattern (Section 5.5) uses a similar centralized technique for assigning tasks to resources across the entire system in a way that facilitates enforcement and analysis of real-time requirements. Both task assignment and then scheduling after the tasks have been assigned are required for predictable real-time enforcement.

5.1 Distributed Scheduling Service

Problem: If globally incompatible policy decisions are made in the local subsystems, the system designer will not be able to predict the real-time performance of the entire distributed application.

Context: Distributed real-time systems where the application has tasks that span operating systems and/or endsystems that have varying scheduling policies, and the designers must be able to predict the real-time behavior of the entire system to some reasonably high level of assurance.

Forces: This pattern is affected by the activities spanning endsystems force, the dynamic heterogeneous applications within an endsystem force, the meta-state consistency force, and the heterogeneity among operating systems' and endsystems' local scheduling force, all described in Section 2.3.

Solution: Provide a scheduling service that application-level tasks use to invoke the scheduling primitives of the underlying operating systems and endsystems in a uniform, predictable, manner.

Resulting Context: All application scheduling-related calls must go through the scheduling service and not be made directly to the OS/endsystem. This allows the scheduling service to provide centralized coordination of all local subsystem policy decisions and resulting subsystem calls in a uniform way that supports predictability. Furthermore, assurances of uniform policies may make forms of real-time analysis possible across the entire systems - thus further enhancing the required predictability.

Rationale: The design emphasizes coordination and of scheduling policies within a single service. The pattern provides uniformity of scheduling decisions within an integrated service interface.

5.2 Distributed Scheduling

Problem: Local subsystem scheduling parameters, such as thread priority, attached to a task that spans subsystems may not be meaningful in the remote subsystem

due to inconsistent scheduling models. For example, in priority-based scheduling, some local operating systems order threads' priorities from low to high, while others order from high to low. These inconsistencies must be reconciled in order to have a coherent global scheduling model.

Context: Distributed real-time systems where the application has tasks that span operating systems and/or endsystems, and the designers must be able to predict the real-time behavior of the entire system to some reasonably high level of assurance .

Forces: This pattern is affected by the activities spanning endsystems force, the dynamic heterogeneous applications within an endsystem force, and the heterogeneity among operating systems' and endsystems' local scheduling force, all described in Section 2.3.

Solution: Provide a coherent set of global scheduling parameters that can map to the specific parameters of the local endsystems and operating systems. For example, in a priority-based system, we could assign priorities to all tasks in the entire system from a single global priority ordering.

Resulting Context: The assignment of global scheduling parameters requires globally consistent knowledge of all tasks in the system (which can be achieved with the Distributed Scheduling Service pattern). Local subsystems require the ability to map these scheduling parameters to their local parameters in a way that supports the required global predictability. For example, the Global to Local Priority Mapping Pattern described in Section 5.4 may be used to implement the Distributed Scheduling pattern: providing a global priority ordering facilitates using the many well-known single-node priority-based enforcement and analysis techniques in the distributed system to increase predictability. The Distributed Scheduling pattern is seen in the Juno **??** meta-programming architecture for heterogeneous scheduling disciplines, and in RT CORBA compliant ORBs such as TAO [31] and ZEN [32].

Rationale: The design *adapts* disparate local parameter value schemes to provide a uniform global view of those parameters. The pattern shields system developers from accidental scheduling complexities introduced by platform and endsystem heterogeneity.

5.3 Distributed Resource Consistency Control

Problem: If incompatible resource access control policy decisions are made in the local subsystems, the system designer will not be able to predict the real-time performance of the entire application or the overall consistency of the resources.

Context: Distributed real-time systems where the application has tasks that span operating systems and/or endsystems that have various local resource access control policies, and the tasks share resources, and the designers must be able to predict the real-time behavior of the entire system to some reasonably high level of assurance.

Forces: This pattern is affected by the activities spanning endsystems force, the dynamic heterogeneous applications within an endsystem force, and the heterogeneity among operating systems' and endsystems' local scheduling force, all described in Section 2.3.

Solution: Require that all tasks in the distributed system access resources using consistent local resource access mechanisms and parameters.

Resulting Context: Ensuring consistent resource access mechanisms and parameters requires coordination across all tasks in the system (which can be achieved with the Distributed Scheduling Service Pattern). This use of global resource access mechanisms and parameters will facilitate calculation of blocking times of the tasks, which in turn facilitates analysis and reasoning about system predictability.

Rationale: The design is based on local coherency of scheduling decisions. The pattern ensures that local consistency enforced coherently across endsystems results in end-to-end consistency.

5.4 Global to Local Priority Mapping

Problem: Global priorities provide a uniform mechanism for ordering all tasks in a real-time system. However, at the OS/endsystem level local priorities may be constrained to smaller ranges than the global priorities. For example, RT CORBA provides 32K global priorities for distributed scheduling. But global tasks must execute on specific operating systems where the number of priorities is smaller (POSIX real-time mandates only 32 local

priorities, for instance). When multiple global priorities are mapped to a single local priority, priority inversion can occur that can compromise the real-time predictability and performance of the overall system.

Context: A distributed real-time system in which tasks are assigned global priorities to support real-time predictability across the entire system, but where the tasks may be executed and enforced on different operating systems/endsystems that have different priority mechanisms.

Forces: This pattern is affected by the activities spanning endsystems force, the dynamic heterogeneous applications within an endsystem force, and the heterogeneity among operating systems' and endsystems' local scheduling force, all described in Section 2.3.

Solution: Map global priorities to local endsystem-level or OS-level priorities by spreading out the tasks on a particular subsystem among the available local priorities. If any tasks with different global priorities must be mapped to the same local priority (e.g. because there are more global tasks than local priorities), then take into account any priority inversion that may occur and use this in any schedulability analysis that is done on the system to support predictability.

Resulting Context: Priority mapping should be done in such as way as to consider future tasks that may arrive. In a dynamic distributed system, tasks must be mapped from global to local priorities as they become available to execute. The priority mapping scheme must consider the overall expected set of tasks to be scheduled (perhaps using probability distributions) so that global priorities are mapped as evenly across the local priorities as possible. Otherwise, a poorly designed mapping algorithm will cause many tasks to be mapped to a very few local priorities.

Rationale: Similar to the Distributed Scheduling pattern, the design *adapts* disparate local priority value schemes to provide a uniform global view of priority. The Global to Local Priority Mapping pattern focuses on priorities, and can be used to implement the Distributed Scheduling pattern.

5.5 Global Load Allocation

Problem: When deciding among several subsystems on which to place the execution of a particular task, certain resources can become overutilized while others may be underutilized. This poor global allocation of resources could cause some tasks to unnecessarily violate timing constraints.

Context: A real-time distributed system in which particular tasks may use any of a set of equivalent resources from one of several operating systems or endsystems.

Forces: This pattern is affected by the multiple capable resources spanning endsystems force, the meta-state consistency force, and the dynamic heterogeneous applications within an endsystem force, both described in Section 2.3.

Solution: Allocate tasks to the subsystems that yield the best chance that the specified timing constraints will be met. Further, consider future tasks when making this allocation. For example, as in classic memory management algorithms, if an task is allocated to the subsystem on which it fits and produces the highest utilization (best fit), when the system becomes highly loaded, there may be small "holes" on the resulting high utilization resources that will not fit any future tasks. Instead, if the task is allocated to the subsystem that will yield the lowest utilization (worst fit), more "medium holes" are left, but one "large hole" gets smaller which reduces the chance of a future large task fitting. The load allocation schemes should choose subsystems with future tasks in mind. This may be done by examining prior distributions of tasks in similar applications, as well as by choosing subsystems on which execution time will likely be freed soon, i.e. subsystems that have tasks ending.

Resulting Context: Load allocation techniques as described above may require some run-time analysis of current system conditions. This will incur added overhead to the execution of the application. For this reason, load allocation algorithms should be designed and implemented carefully to utilize as much precomputed system information as possible, and avoid unnecessary analysis. Alternatively, simple load allocation techniques (like first fit, or simple balancing algorithms) may be sufficient and would incur less overhead.

Rationale: The design emphasizes load distribution to improve overall distributed system behavior. The pattern balances overhead of analysis with expected future behavior of the system and potential improvements in system performance.

5.6 Global Overload Management

Problem: In a dynamic distributed real-time system, system load can vary, sometimes exceeding the capacity of available resources. When there is not enough capacity to handle all execution, the system must be able to ensure that critical tasks meet their specified timing constraints. The initial allocation of load (specified in Global Load Allocation Pattern of Section 5.5) attempts to place load on subsystems so that all timing constraints can be met. The techniques involved in load allocation typically use a "best-guess" at where each task execution will fit, while leaving room for future execution. However, as time progresses, these guesses are replaced by real incoming task demands, which may not meet the expectations of the original load allocation.

Context: Distributed real-time systems where the application has tasks that span operating systems and/or endsystems, and some application tasks may be less Critical than others. That is, there is a possibility that application tasks may be re-allocated, reduced or shed altogether if system conditions require it and Critical tasks should be completed when possible.

Forces: This pattern is affected by the multiple capable resources spanning endsystems force, the dynamic heterogeneous applications within an endsystem force, the meta-state consistency force, and the competing Quality of Service requirements force, all described in Section 2.3.

Solution: Provide distributed middleware-level QoS adjustment that includes monitoring and managing system load to avoid overload on the entire system and on each particular subsystem in a way that allows the most Critical tasks to meet their timing constraints. The monitoring will require an overview of what tasks are executing where, and how each resource is being utilized, and the completion status of tasks. The management of load may include reallocation of tasks from one subsystem to another to avoid overload. It may also include reducing the execution time (QoS Accuracy), or shedding the execution altogether of less critical task to ensure that the most critical tasks meet their timing constraints.

Resulting Context: Management of load in a distributed system will require a global view of the status of the system at any given moment. The level of overhead involved in this management depends upon the granularity of information that is accessible, and on how often this information is needed. The strategies chosen for overload management should weigh the benefits of having access to detailed information with the time needed to analyze it. Shedding of tasks must be supported in the underlying endsystem and operating system such that the tasks can be shed gracefully and efficiently without causing inconsistent system state.

Rationale: The design emphasizes distributed load balancing and reallocation to tune overall distributed system behavior at run-time. The pattern balances overhead of load re-allocation with expected improvements in system performance.

6 Rationalizing Scheduling Across Levels of Scale

Finally, we present two patterns that apply across levels of abstraction. The Application Level Quality of Service Adjustment pattern (Section 6.1) allows applications to themselves adjust their scheduling parameters to facilitate better global enforcement of real-time requirements. The Distributed Temporal Coherency pattern (Section 6.1) ensures appropriate end-to-end semantics for reasoning about and enforcing time, which are needed by the other patterns in this section.

Implementations of either of these patterns may appear at multiple levels of abstraction, or may in fact span levels. For example, in a video streaming application, an application may make network bandwidth reservations at the OS level, a feasible priority assignment on one or more endsystems. If due to, e.g., the arrival of more critical processing, these reservations become infeasible, the Application Level Quality of Service Adjustment pattern may be applied to re-negotiate or adapt to (1) the new OS level bandwidth available, (2) the new endsystem priority as-

signment, or (3) a combination of the two, with respect to the particular application semantics.

6.1 Application Level Quality of Service Adjustment

Problem: In many applications QoS parameters such as timing constraints, accuracy, and security parameters are specified once and the system decides whether or not they can be achieved. When an application cannot receive all of its requested quality, there should be a way for the application itself to negotiate a trade-off that is acceptable to it, while still meeting the constraints of the system.

Context: Systems in which quality of service (QoS) negotiation among applications and implementation infrastructure is allowed, and overload may occur.

Forces: This pattern is affected by the competing Quality of Service requirements force described in Section 2.3 and the fact that applications often have more information about the best way to provide what they need than the overall system.

Solution: Provide a mechanism that allows applications to specify what they are willing to sacrifice in order to meet their own constraints as well as the overall system constraints. For example, QuO [7] provides contracts that specify how to reduce certain demands of an application when the system becomes overloaded. In [33], a real-time agent specifies several execution strategies to provide a particular service with varying levels of execution time and resulting quality. When a request for this service is received, the agent negotiates with the scheduling service to determine the quality it can provide.

Resulting Context: Allowing applications to have some control over how their QoS is provided will enhance the overall quality of the system. It will require application programmers to provide more information about how the application can be run, and about what they are willing to sacrifice to meet the overall goals of the application and the system.

Rationale: The design emphasizes application-level flexibility in adapting to QoS limitations at run-time. The pattern rationalizes alternatives for re-establishing feasible and acceptable QoS.

6.2 Distributed Temporal Coherency

Problem: Insufficiently fine-grained resolution in either (1) the timeliness information available, or (2) enforcement capabilities of mechanisms using that information, may limit the precision at which timeliness assurances can be made.

Context: Application timeliness requirements induce particular time-scales at which temporal enforcement must be possible within operating systems, endsystems, and distributed systems.

Forces: The temporal consistency force is a special case of the meta-state consistency force that is essential to determine end-to-end feasibility of timeliness requirements, and to rationalize resource allocations across all levels of scale. Different forms of temporal consistency can be useful, such as synchronization of clock periods versus synchronization of clock values. Furthermore, OS-level algorithms for clock synchronization must maintain consistency within some bound on precision needed by the distributed services to assure end-to-end properties. Applications with fine-grained timeliness requirements may need increased resolution of timing information and enforcement mechanisms. In a distributed system, expense increases with finer granularity of time consistency, up to a possible point of diminishing returns dictated by the economics of the system. Furthermore, the achievable resolution is ultimately bounded within strict physical limitations.

Solution: Provide time consistency that is sufficient to meet application requirements, in the most economical manner. This applies both to clock synchronization and other time-based factors such as the latency across a network link.

Resulting Context: An application may reach either the economic or physical bound first. For example in switched ethernet where collisions are avoided, throughput is bounded by the transmission properties of the medium and the buffer capabilities on the endsystem. While the choice of buffer sizes in a switch may be dictated by economic factors, the network medium transmission characteristics are determined by physics.

Rationale: The design deals rationally with inherent limitations on the systems ability to attain or justify increasingly fine granularity of temporal coherency. The

pattern offers a balance between inherent requirements and the costs of alternatives to meet those requirements.

7 Conclusions

The patterns described in this paper apply both within and across different levels of abstraction in a distributed system. Ultimately, they are about maintaining timeliness constraints in distributed real-time systems, and do so by applying different scheduling patterns and supporting patterns to achieve rational allocations of resources across the distributed system. The design choices made to resolve forces at one level of abstraction may induce consequences that are themselves forces at another level. Furthermore, some patterns apply across levels of abstraction, resolving forces that are within or possible across the levels themselves.

The key contribution of this pattern language is its exploration of the complex and interconnected design space for scheduling distributed real-time applications. It is capable of generating designs that adhere entirely to one design paradigm, e.g., preemptive priority-based thread scheduling or planned time synchronized scheduling. It is also capable of generating designs that compose elements of multiple paradigms, thus offering the designer freedom to hybridize approaches as long as the design forces are suitably resolved.

8 Thanks

We wish to thank our Patterns Shepherd, Angelo Corsaro, for his insightful comments and suggestions for improvement of this paper.

References

[1] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[2] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[4] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[5] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[6] C. D. Gill, R. Cytron, and D. C. Schmidt, "Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems," in *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.

[7] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[8] O. Uvarov, "Dynamic real-time scheduling and load shedding for qos middleware," TR TR02-286, University of Rhode Island, 2002.

[9] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 166–171, IEEE Computer Society Press, 1989.

[10] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[11] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization," *IEEE Transactions on Computers*, vol. 39, September 1990.

[12] D. Zhu, R. Melhem, and B. Childers, "Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems," in *The 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, (London UK), December 2001.

[13] A. Corsaro, D. C. Schmidt, R. K. Cytron, and C. Gill, "Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications.*, (Rome, Italy), pp. 289–299, OMG, September 2001.

[14] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications ACM*, vol. 26, no. 7, pp. 558–565, 1978.

[15] H. Kopetz, "The Time-Triggered Model of Computation," in *The 19th IEEE Real-Time Systems Symposium (RTSS '98)*, (Madrid Spain), Dec. 1998.

[16] R. Gerber, W. Pugh, and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks," *IEEE Transactions on Computers*, vol. 44, Mar. 1995.

[17] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[18] M. Saksena, J. da Silva, and K. Agrawala, "Design and Implementation of Maruti-II," in *Advances in Real-Time Systems* (S. Son, ed.), Prentice Hall, 1995.

[19] D. A. Solomon, *Inside Windows NT, 2nd Ed.* Redmond, Washington: Microsoft Press, 2nd ed., 1998.

[20] IBM, "OS/2 Warp." `www-3.ibm.com/software/os/warp/`, 2002.

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

[22] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, vol. 8, pp. 62–72, May 1991.

[23] Y.-C. Wang and K.-J. Lin, "Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *IEEE Real-Time Systems Symposium*, pp. 246–255, IEEE, December 1999.

[24] D. Lea, *Concurrent Java: Design Principles and Patterns, Second Edition*. Reading, Massachusetts: Addison-Wesley, 1999.

[25] U. C. Guard, "Global Positioning System Standard Positioning Service Specification, 2nd edition." `www.navcen.uscg.mil`, 1995.

[26] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[27] I. Pyarali, D. C. Schmidt, and R. Cytron, "Achieving End-to-End Predictability of the TAO Real-time CORBA ORB," in 8^{th} *IEEE Real-Time Technology and Applications Symposium*, (San Jose), IEEE, Sept. 2002.

[28] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[29] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in *Real-Time Programming* (W. Halang and K. Ramamritham, eds.), Tarrytown, NY: Pergamon Press, 1992.

[30] J.-Y. Chung, J. W.-S. Liu, and K.-J. Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," *IEEE Transactions on Computers*, vol. 39, pp. 1156–1174, September 1990.

[31] Center for Distributed Object Computing, "The ACE ORB (TAO)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[32] Center for Distributed Object Computing, "The ZEN ORB." www.zen.uci.edu, University of California at Irvine.

[33] L. C. DiPippo, V. F. Wolfe, L. Nair, E. Hodys, and O. Uvarov, "A Real-Time Multi-Agent System Architecture for E-Commerce Applications," in *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems*, Mar. 2001.