

# Integrated CORBA Scheduling and Resource Management for Distributed Real-Time Embedded Systems

Kevin Bryan, Lisa C. DiPippo, Victor Fay-Wolfe,  
Matthew Murphy, and Jiangyin Zhang  
Department of Computer Science  
University of Rhode Island  
Kingston, Rhode Island 02881, USA  
{bryank,dipippo,wolfe,murphym,zhang}@cs.uri.edu

Douglas Niehaus  
Electrical Engineering & Computer Science Department  
University of Kansas  
Lawrence, Kansas 66045, USA  
niehaus@eecs.ukans.edu

David T. Fleeman, David W. Juedes, Chang Liu,  
and Lonnie R. Welch  
School of Electrical Engineering and Computer Science  
Ohio University  
Athens, Ohio 45701, USA  
{fleeman,juedes,liuc,welch}@ohio.edu

Christopher D. Gill  
Department of Computer Science and Engineering  
Washington University  
St. Louis, Missouri 63130, USA  
cdgill@cse.wustl.edu

**Keywords:** QoS in open systems; integrated scheduling and resource management; CORBA

## Abstract

*Integration of middleware scheduling and resource management services enables open distributed real-time embedded (DRE) applications to meet end-to-end quality of service (QoS) requirements in highly variable operating environments. This paper describes our research<sup>1</sup> on integrating CORBA scheduling and resource management services, and presents experiments we conducted to validate and quantify the benefits of this integration.*

*Our experimental results show that integrating distributed scheduling and resource management in middleware for open DRE systems can offer significant improvements in predictability. Specifically, integrating our stand-alone resource management service with a previously unmanaged experimental baseline application reduced the ratio of missed deadlines from 26% to 10%, and the same application performed even better under the control of integrated scheduling and resource management services, with a missed deadline ratio of only 1%.*

## 1. Introduction

Quality of service (QoS) management in open distributed real-time and embedded (DRE) systems is particularly challenging because open DRE systems cannot assume upper bounds on resource demands by system tasks, or lower bounds on resource availability that are the norm in closed DRE systems. Task resource demands in an open

DRE system may exceed any particular static allocation of available resources, due to changing environmental conditions, damage to or destruction of resources, and/or evolving application objectives.

**Motivating applications:** Open DRE systems in which resource demands may exceed available resources include military command and control [11], robotic search and rescue [35], and adaptive audio/video streaming [16].

For example, in a time-sensitive military mission replanning application, a command-and-control aircraft operator can collaborate with the weapon systems officer (WSO) in a strike aircraft by relaying tasking orders and imagery, and interacting with the WSO to re-plan the aircraft's current mission to engage a time-critical target [11]. In addition to each aircraft's own critical processing requirements, it may be necessary to ensure that elements of the collaboration session between the aircraft also receive critical assurances, e.g., for time-bounded delivery of collaboration alerts and new tasking orders.

**QoS management requirements:** In general, open DRE systems require four key resource management capabilities:

1. Assurance of timeliness and other QoS requirements along end-to-end paths that may span only endsystems connected by a common backplane, or may span dispersed endsystems across multiple intervening hops;
2. Custom configurability of middleware policies and mechanisms for efficiency and improved assurance of meeting timeliness and other QoS requirements;
3. Timely adaptation within rapidly changing operating environments of open systems; and
4. Opportunistic optimizations of QoS based on situational factors.

**Limitations of previous approaches:** These requirements span classical DRE system boundaries, including endsystems, architectural layers, and service interfaces. Historically, middleware resource management approaches

---

<sup>1</sup> This research was supported in part by the DARPA PCES program, contract # F33615-03-C-4111.

(e.g., [16][18][21][25]) have produced layered solutions in which services independently mediate interactions between the application and lower-level system software. These approaches have key shortcomings for *open* DRE systems in which end-to-end QoS requirements must be assured, configured, adapted, and optimized within highly variable operating environments. This is because coarse-grained interactions between independent services cannot provide sufficient and timely information for the services to be able to make sound decisions within fine-grained time scales.

**Solution approach → integrated resource management in middleware:** To address the above challenges for open systems like the military mission replanning example, we have developed an integrated approach based on our previous work on scheduling patterns [10][13]. Our previous research efforts have examined the capabilities of individual resource management services, and have made preliminary investigations of the benefits of cross-cutting the barriers between individual layers and services, such as integrating scheduling services with higher-level QoS management services for improved predictability of real-time image transmission [11].

This paper describes our work on developing global, distributed, and integrated scheduling and resource management capabilities, which extends our earlier research to address the integration of peer services within the same middleware QoS management layer. Our objective in this research is to provide DRE applications better QoS enforcement, through careful integration of policies and mechanisms for how resources are allocated and how access to those resources is arbitrated [4]. This finer-grain integration of services can enable better decision making *across* services without sacrificing decoupled, component-based development of individual services, which is also important for open systems.

Our integrated approach consists of a *distributed scheduling* (DS) service [4], which coordinates the activities of multiple endsystems, a *distributed resource management* (DRM) service [6], which allocates resources on each endsystem, and several alternative endsystem scheduling options, e.g., Real-Time CORBA 1.2 fixed priority schedulers [17], Kokyu [14], and *group scheduling* [8]. We have integrated our DS and DRM services atop the TAO [30] CORBA [24] object request broker, and have performed experiments to validate and evaluate this integration, which we present in Section 4.

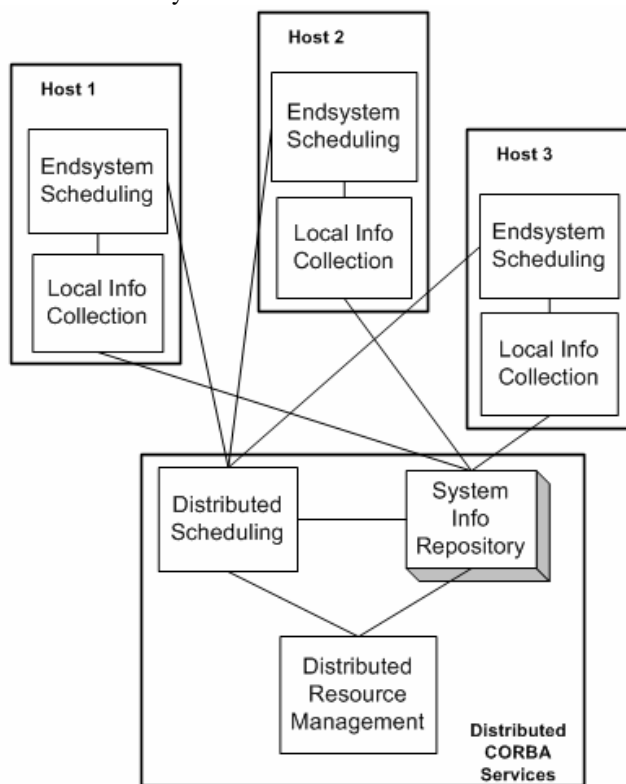
The system architecture of our approach is presented in Section 2. Section 3 presents the integration of two top-level components in this architecture, the distributed scheduling service and resource management services, and outlines next steps for integration with a wider range of endsystem schedulers. Section 4 describes and presents the results of experiments that we performed to validate and evaluate our approach. Section 5 discusses related work. Finally, Section 6 offers concluding remarks.

## 2. System Architecture

The key to our approach is a system architecture that enables integration of the following focus areas:

- Distributed Scheduling
- Distributed Resource Management
- Endsysteem Scheduling
- Information Collection and Use

Figure 1 illustrates the architecture of an open DRE system involving three endsystems under the control of our integrated scheduling and resource management capabilities. Figure 1 shows how the four areas are reflected in the system architecture.



**Figure 1. Integrated QoS Management Architecture**

On each endsystem, a local scheduler arbitrates access to resources within that endsystem, and a local information collection component records a variety of status information such as CPU utilization, progress of application activities, and success or failure of tasks in meeting their deadlines. This local status information is distilled into higher-level information, such as predictability of local tasks in meeting intermediate deadlines toward timely completion of end-to-end activities. The higher-level information is sent to a distributed information collection service called the *system information repository*. In addition to the local scheduling and local and distributed system repository components,

there are two other distributed CORBA services in our QoS management architecture, the distributed scheduling service and the distributed resource management service. These services work together to provide fine-grain adaptive scheduling and resource management capabilities to the open DRE system.

By adopting this architecture, we aim to improve system quality, reduce end-to-end latency, and provide rigorous management of scheduling aspects across systems. The four focus areas are further explained in the following subsections.

## 2.1. Distributed Scheduling

We have developed a distributed scheduling service [4] to provide distributed real-time scheduling decisions for local enforcement in the TAO middleware [30]. The DS service is informed at system initiation of relevant local scheduling attributes for individual endsystem scheduling points along end-to-end paths. Such attributes may include local operating systems, event queues, POA (Portable Object Adapter) policies, dispatchers, and network routers. For instance, it will be told if a Real-Time CORBA 1.2 (RTC 1.2)<sup>2</sup> [25] endsystem scheduler enforces a fixed priority or earliest deadline first (EDF) policy. The DS service is also made aware of the end-to-end activities that must be supported in the middleware. If given global knowledge of all end-to-end executions and of all local scheduling capabilities, the DS service can set globally-sound scheduling parameters for each local enforcement mechanism. For instance, if the DS service knows that all endsystem schedulers are RTC 1.2 compliant and enforce EDF, the DS service will determine a globally-sound deadline for RTC 1.2 distributable threads at each of the scheduling points each distributable thread will encounter.

Our DS service implementation uses the Interceptor pattern to place invocation hooks within “wrappers” for the local enforcement mechanisms. The wrappers then intercept calls to the local enforcement mechanisms, and interact with the DS service through those hooks. This interaction may be a local call to a DS service daemon on the end-system, or may require a call to a remote component of the DS service. For instance, if a RTC 1.2 distributable thread sets its deadline, the wrapper for the local RTC 1.2 scheduler will first check with the DS service to determine if the requested deadline should be altered. Furthermore, subsequent dispatches of subtasks in the distributable thread will check with the DS service to allow intermediate deadlines to be set.

In the implementation with which the experiments presented in Section 4 were conducted, both distributed

scheduling and distributed resource management services have been integrated, and local enforcement is done by TAO’s RTC 1.2 fixed priority scheduling service [17]. We have also integrated a distributed dynamic scheduling service with Kokyu [9][12][14] based endsystem scheduling. Kokyu can be configured to support many local enforcement mechanisms. Our initial distributed dynamic scheduling approach uses Sun’s algorithms [32], which establish release times and intermediate deadlines for subtasks in an end-to-end task. Since Sun’s algorithms assume EDF scheduling on local operating systems, we have configured Kokyu to do EDF scheduling.

To support enforcement of a wide range of application-customized distributed scheduling policies, such as ensuring consistent progress of multiple end-to-end activities, we have implemented and evaluated an endsystem scheduling approach called group scheduling [1] which is described in Section 2.3. As future work we will integrate distributed dynamic scheduling with our progress-based group scheduling approach. We will also integrate our distributed resource management service and distributed dynamic scheduling service for both the Kokyu and group scheduling endsystem scheduling variations, to compare and contrast the effects of different endsystem scheduling approaches on distributed scheduling and resource management.

## 2.2. Distributed Resource Management

We have implemented a distributed resource management service in CORBA as part of the Quality-based Adaptive Resource Management Architecture (QARMA) [6] project. The QARMA DRM service consists of the following types of elements:

- *monitors* that gather information about resource usage, resource availability, application performance, and environmental conditions;
- *detectors* that evaluate particular subsets of the information in the system repository, to decide whether or not to trigger the decision-maker to perform a reallocation;
- a *decision-maker* that uses a subset of the information in the system repository to decide what actions should be performed to ensure that end-to-end performance requirements are satisfied and that overall end-to-end performance is improved when opportunities arise; and
- *enactors* that receive instructions from the decision-maker about what actions to perform in the system, and then enact those actions.

Different types of monitors serve different purposes, including host monitors that watch resource usage on a single host computer, and path monitors that watch events that take place along a particular end-to-end path [25].

---

<sup>2</sup> The OMG has recently renumbered the second Real-Time CORBA specification from 2.0 to 1.2. The contents of the specification remain the same, however.

### 2.3. Endsystem scheduling

At each endsystem, access to resources by competing activities must be scheduled according to end-to-end application level requirements. In our architecture, the DS service translates those end-to-end requirements into local resource scheduling policies enforced on each individual endsystem. Resource scheduling mechanisms used to enforce those local policies can be provided at both the middleware and operating system levels. Historically, such scheduling mechanisms have included:

- ordered queues in the Kokyu framework and thread pools with lanes in TAO's Real-Time CORBA 1.0 implementation [26] at the middleware level, and
- low-level abstractions such as prioritized or share-based allocation of CPU cycles in KURT-Linux [23][31] or TimeSys Linux [33], or network bandwidth in RSVP [3] at the operating system level.

Although these traditional mechanisms are sufficiently powerful to express the resource allocation requirements of distributed real-time applications, programming those abstractions directly can be both tedious and error prone for even moderately complex systems. Our previous research has shown that higher-level application progress requirements can be mapped to scheduling enforcement mechanisms more directly and efficiently, through modular intermediate abstractions provided by an approach we call *group scheduling* [8].

Our group scheduling model combines the advantages of two familiar scheduling paradigms: hierarchical [15][28][29] and path-oriented [1] [20]. Under recent modifications to KURT-Linux, the operating system (OS) implementation of the group scheduling model can control execution of every type of computation component under Linux, including threads, interrupt handlers, soft-IRQs, bottom-halves, and tasklets [7]. The group scheduling model has also been implemented at the middleware level [8], and our recent empirical evaluations [1] show qualitatively similar performance between the OS and middleware implementations, albeit at a cost of additional overhead for the group scheduling mechanisms in middleware.

We have integrated traditional endsystem scheduling frameworks such as TAO's RTC 1.2 fixed priority scheduler and the Kokyu framework with our distributed scheduling service. We will also integrate our group scheduling framework with distributed scheduling and resource management, to allow finer-grained enforcement of end-to-end activities progress requirements. To facilitate that integration, we have developed wrapper façades for the middleware and OS group scheduling interfaces, in the ACE framework upon which TAO and our DS and DRM services are built.

### 2.4. Information collection and use

The distributed resource management (DRM) subsystem is responsible for ensuring that adequate resources are available for use when and where they are required along the end-to-end path of a computation through the distributed system. Precisely what kind of information is required depends on the computations being supported, but (1) the DRM subsystem must have access to a wide variety of information and (2) the precise set of information required will vary with system configuration and operating conditions. The set of data collection requirements is complicated further by the needs of endsystem scheduling, and our need to evaluate the performance of our end-to-end distributed computations. The combined set of requirements is significantly more demanding than commonly used performance data collection methods can handle, and has required considerable effort on our part in support of data collection.

Our endsystem data collection system, called Data Streams, supports data gathering from both the application level through its Data Streams User-level Interface (DSUI), and from the underlying endsystem OS through its Data Streams Kernel Interface (DSKI) [22]. The DSUI is portable across a wide range of endsystems, requiring access only to an endsystem time standard for time-stamping data records. The DSKI is a standard device driver and is thus also reasonably portable, though creation of an instrumentation point requires access to the source code being instrumented. The Data Streams subsystem supports name spaces of instrumentation points that generate a performance datum when a thread of control passes through an instrumentation point in application, middleware, or OS code. Each instrumentation point is disabled by default, and can be enabled, configured, and associated with a data stream selectively as required for each task.

Data from each instrumentation point is in a standard format permitting sets of data gathered separately to be merged and processed in a number of ways. This ability to merge data gathered separately is important because the set of information required to answer a particular performance evaluation question, to support a dynamic scheduling algorithm on-line, or to support the DRM subsystem may come from many sections of the system. The Data Streams subsystem is thus aspect-based in the sense that it supports configuring the data set collected for a particular situation, regardless of how the sources of that data cut across conventional system component boundaries.

The QARMA DRM service monitors, detectors, decision-makers, and enactors also make use of the system repository. For example, the monitors store the data they gather into the system repository; the detectors and the decision-makers use data in the system repository to make

their decisions. The information in the system repository is also accessible to the DS service and the local endsystem schedulers. The system repository stores information about which end-to-end tasks are currently running, and on what resources. This will enable, for example, the DS service to determine if a new distributable thread can be scheduled along with the other tasks currently running in the system. Such sharing facilitates finer-grain integration of the DS and DRM services, and the local endsystem schedulers.

### 3. System Integration

We are approaching the problem of system integration in an incremental fashion. First, we have integrated the DS service and the DRM service, as this paper describes. These services currently are integrated with a basic RTC 1.2 fixed priority scheduler on the local endsystems.

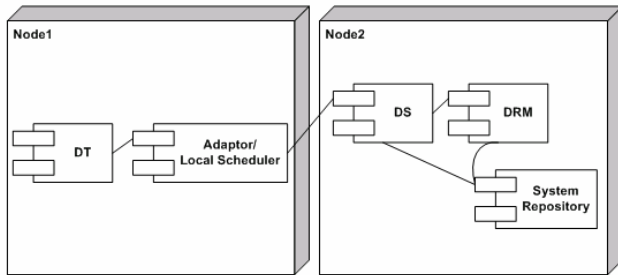


Figure 2. Integrated DS and DRM Services

**Integration of the DS and DRM services:** The integration of the DS service with the QARMA Resource Manager [6], and with TAO’s RTC 1.2 scheduling service, is shown in Figure 2. In the interactions depicted in Figure 3, the application makes a *begin\_scheduling\_segment()* call (labeled *bss* in the figure) to a distributable thread (labeled *DT*). The distributable thread then makes a *begin\_new\_scheduling\_segment()* call (labeled *bnss*), which is intercepted by the DS service wrapper. The wrapper sends the requested scheduling parameters to the DS service via a *begin\_distributed\_scheduling\_segment()* call (labeled *bdss*). The DS service performs a global schedulability analysis (currently rate monotonic utilization bound analysis on all nodes).

If the system is schedulable, the DS service sets the scheduling parameters (currently the priority) for the distributable thread on the local enforcement mechanism(s) (TAO’s RTC 1.2 scheduler). If the system is not schedulable, the DS service makes a *rm\_reactive()* call and invokes the DRM service, which either adjusts the QoS of the execution or reallocates resources. The system repository, as depicted in the system architecture, is the central storage of all static and dynamic configuration and status information for the application.

Both the DRM service and the DS service share the system repository, so they maintain a common picture of resource status and usage in the entire system. The experiment results presented in Section 4 validate the integration of the DS service and the DRM service.

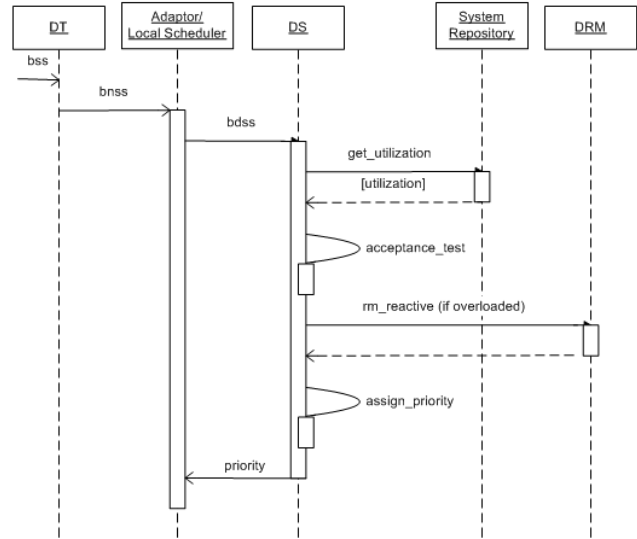


Figure 3. Interactions between DS and DRM Services

**Further system integration:** In addition to our integration of the DS and DRM services with a RTC 1.2 fixed priority scheduler, we have also integrated a distributed dynamic scheduling service with the Kokyu endsystem scheduling framework, and will subsequently integrate the DRM service with that instantiation of our architecture. As future work, we will upgrade that Kokyu-based integration to use group scheduling capabilities.

The results of our endsystem group scheduling experiments [1] show that both priority and share based scheduling policies can be combined with different interpretations of fairness, which in the context of distributed scheduling and resource management will allow us to enforce a wider range of effective end-to-end scheduling policies. It is important to note that our group scheduling approach could be implemented directly within Kokyu for applications whose progress semantics match EDF-style enforcement of sub-task deadlines on each endsystem along an end-to-end path.

This observation offers a potential migration path for a distributed scheduling service from a traditional Kokyu endsystem scheduler to a Kokyu-based implementation of group scheduling policies, and finally to the direct group scheduling implementation [1]. With these alternatives in mind, we will conduct further experiments to evaluate our approach as we begin final integration of our resource management, distributed scheduling, and local endsystem scheduling service implementations.

## 4. Experiments

We conducted a set of experiments to validate our approach of integrating resource management and distributed scheduling. We first measured the performance of a DRE system controlled by the QARMA DRM service alone. We then measured performance of the same DRE system controlled by the joint DS and DRM services. We also compared those results to the performance of the same DRE system without either DS or DRM services.

### 4.1 Experiment setup

Our experiments were based on a distributed video delivery application for Unmanned Aerial Vehicles (UAVs). The UAV application consists of *senders*, which acquire and send out video frames; *viewers*, which either display the video to human monitors or feed the video to automatic target recognizers; and *distributors*, which replay the videos frames from senders to viewers. The DRM service had been integrated with the UAV application in previous work [6].

In our experiments, we collected data from a simulated UAV application that runs in RMBench, a tool designed for performance evaluation of real-time embedded systems [5]. We used the simulated UAV for these experiments because the real UAV application is maintained by a third party and has only three dramatically different service levels, in frames per second (FPS): 30 FPS, 10 FPS, 2 FPS. Use of the simulated UAV environment in our experiments allowed us to use ten service levels and to demonstrate more clearly the benefits of our approach in a single set of experiments.

The experiments were performed in the Emulab [34]. We used 4 PCs with single 850MHz CPUs running Red Hat Linux 9. The PCs were connected by a 100 Mbps network. One PC hosted all senders in the UAV application; the second hosted all distributors; the third all viewers; and the fourth the DS and DRM services.

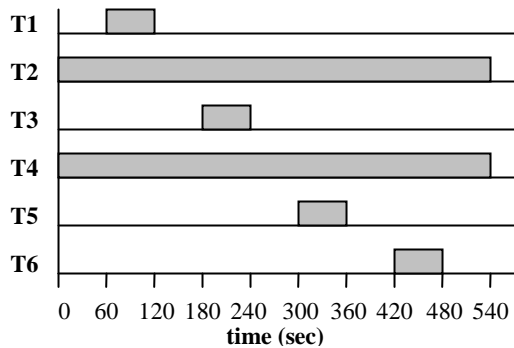


Figure 4. Experiment Scenario with Six Tasks.

The experiment consisted of six periodic end-to-end tasks, which in this case were sender/distributor/viewer chains. Task 1 through Task 6 were assigned importance 1 through 6 (1= lowest, 6 = highest), respectively. Each task consisted of three sub-tasks: sender, distributor, and viewer. Each task could be run at 10 different service levels, with level 10 providing the highest video quality and level 1 the lowest. Task 2 and task 4 ran for the entirety of the experiment, which was 9 minutes, and were initially schedulable on each of the nodes at service levels 5 and 10 respectively. Task 1 then entered the system and caused overload on some of the nodes. Task 1 then left the system and task 3 entered, again overloading the system. Tasks 5 and 6 then also entered the system at different times. Task properties are summarized in Table 1. The key time periods in the experiment are summarized in Table 2. The timeline of these events is depicted in Figure 4.

Table 1. Task Properties in the Experiments.

Task	Task Importance	Period (sec)	Worst Case Execution Time (sec)
T1	1	3	1.3
T2	2	3	1.3
T3	3	3	1.3
T4	4	3	1.3
T5	5	3	1.3
T6	6	3	1.5

Table 2. Timeline for the Experiments.

Time (sec)	Action	Result At Max Quality
0	T2 and T4 begin	Schedulable
60	T1 begins	Not Schedulable
120	T1 ends	Schedulable
180	T3 begins	Not Schedulable
240	T3 ends	Schedulable
300	T5 begins	Not Schedulable
360	T5 ends	Schedulable
420	T6 begins	Not Schedulable
480	T6 ends	Schedulable
540	T2 and T4 ends	Not Schedulable

We ran three instances of the experiment. The *baseline* experiment did not use the DRM service or the DS service to control timeliness. The *DRM service* experiment used only the DRM service, which reacted to host overload. The *DS / DRM service* experiment used the integrated DRM and DS services.

Specific metrics were developed for these experiments to measure the overall system value achieved for each test, and the overhead of both the DS service and DRM service components. Traditional metrics such as end-to-end latencies and statistics describing the quality settings during the experiment were also collected.

## 4.2 Experiment results

To compare the relative values of using the DRM service and the integrated DRM and DS services, the same metrics were collected for all three experiments: *performance metrics* and *overhead metrics*. *Performance metrics* indicate how well the managed system performed in each experiment and include *latency*, *number of missed deadlines*, and *quality* (i.e., the level of service). Any latency larger than three seconds indicates a missed deadline. *Overhead metrics* assess the overhead introduced by the management components. The latencies of the DS service, DRM service, and supporting QARMA management components were measured to show the *response time* from overload detection to enacting changes in the operation of the system to restore feasibility.

### 4.2.1 Performance Metrics

Table 3 summarizes the overall metrics for each stream over the duration of the entire experiment for all three experiments. The results indicate that the use of the management components improve both the *latency* and the *number of missed deadlines* significantly for this experiment, but at the cost of reducing the average quality of some of the distributed tasks.

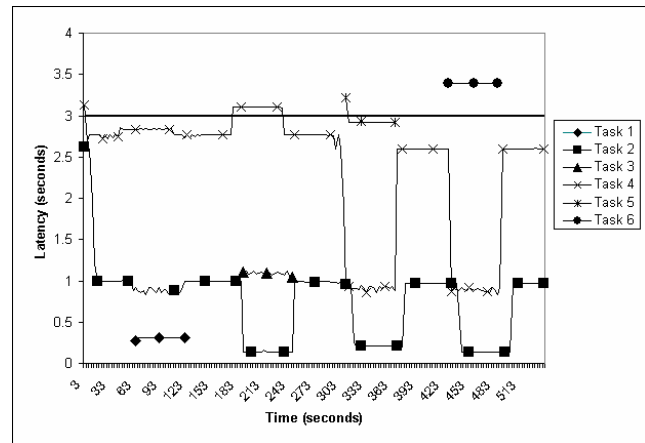
**Table 3. Metrics.**

	Avg. Latency (sec)	Max. Latency (sec)	Number of Missed Deadlines	Avg. Quality
<i>Baseline</i>				
Task1	4.05869	4.27175	20	10
Task2	2.68714	5.61852	32	10
Task3	2.58997	2.71341	0	10
Task4	2.68408	5.58821	44	10
Task5	2.46839	2.87625	0	10
Task6	9.13197	10.3683	15	10
<i>DRM service</i>				
Task1	0.235812	0.247172	0	1
Task2	0.720078	2.606511	0	3.6
Task3	0.984620	1.019679	0	4
Task4	2.367169	3.085342	21	8.5
Task5	2.910771	3.171827	1	10
Task6	3.367901	3.385024	20	10
<i>DS / DRM service</i>				
Task1	0.44564	1.41718	0	1
Task2	1.22624	4.15631	2	3.6
Task3	1.54162	2.7224	0	4
Task4	2.24104	3.11162	1	8.9
Task5	2.6013	2.97634	0	10
Task6	3.01883	3.83688	1	10

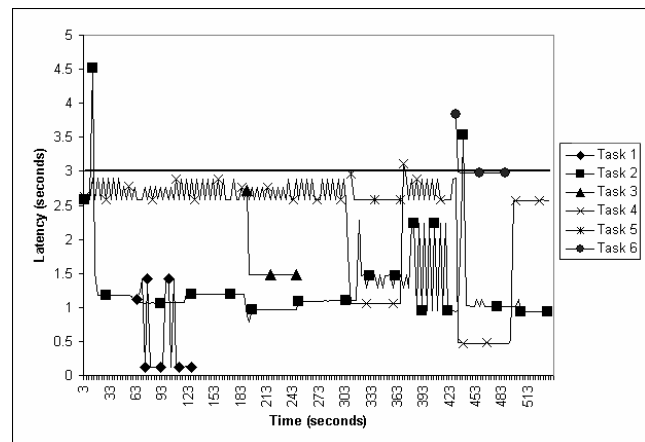
Using the DRM service alone decreased the number of missed deadlines from 26% to 10%. Note that the DRM Service does not set task priorities, so there is no guarantee that with CPU contention and using the DRM service alone, the most important tasks (i.e., tasks 5 and 6) will have

sufficient CPU time. The integrated DS service and DRM service reduced the number of missed deadlines to less than 1%. Since the DS service is able to control local priorities, the *DS / DRM service* experiment showed improvements in both latency and number of missed deadlines over the *DRM service* experiment where local priorities could not be controlled.

The average latency went from 3.93671 seconds in the baseline to 1.76438 seconds in the *DRM service* experiment and to 1.9752 seconds in the *DS / DRM service* experiment. Figures 5 and 6 show the average response times in the *DRM service* and the *DS / DRM service* experiments, respectively<sup>3</sup>. Latencies greater than three seconds constitute a missed deadline. Execution times for each task were always fixed as worst-case execution times. Missed deadlines in the *DS / DRM service* experiment only occurred when a new task entered the system, and before the DRM service had the opportunity to change the service levels of the appropriate task.



**Figure 5. DRM Service Response Times.**



**Figure 6. DS / DRM Service Response Times.**

<sup>3</sup> Data density in Figures 5, 6, and 7 has been reduced to improve readability.

Another important observation from the data is that the integrated DS and DRM services were able to degrade the performance of less important tasks, allowing the more important tasks to run at higher quality levels. Figure 7 shows the quality of the applications over time for the *DS / DRM service* experiment. When task 5 entered the system, the less important tasks 2 and 4 were degraded to allow task 5 to execute at the highest quality setting. When task 5 exited the system, tasks 2 and 4 were returned to higher quality settings to consume the resources released by task 5.

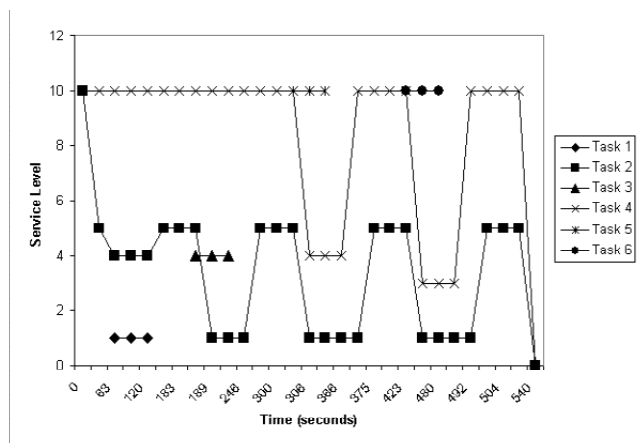


Figure 7. *DS / DRM Service Experiment Service Levels.*

Note that in the *DS / DRM service* experiment, four tasks missed deadlines. All of these missed deadlines occurred immediately after a task that would make the system unschedulable entered the system. These deadline violations occurred because in the RMBench architecture, the enactor must wait for the task to finish the current periodic workload before it can change the service level of the task. In effect, the node remained overloaded until the node finished execution for particular period. No other tasks missed deadlines once the DRM service was able to change the service level of the least important active task.

#### 4.2.2 Overhead Metrics

The elapsed time from overload detection to the notification of every application of new quality levels is referred to as the *management response time* (MRT). In both the *DRM service* and the *DS / DRM service* experiments, the MRT was measured in the same manner, but the overall process was initiated differently. In the *DRM service* experiment, overload detection was accomplished using simple host utilization detectors. These detectors invoked the DRM service whenever a host was either in *overload* or in *underload* as defined by a utilization threshold. In the *DS / DRM service* experiment, overload detection occurred within the DS service, which invoked the *DRM service* if overload was detected or if a task had left the system. In both scenarios, the DRM service computed service level changes and invoked the

QARMA Enactor Service to carry out those changes. The Enactor Service then invoked lower level enactors that updated the service level for each task in the system.

In the *DRM service* experiment, twelve application state changes were identified, each resulting in a call to the DRM service. The average MRT produced by these calls was 0.13697 seconds, and the maximum was 0.16748 seconds. In the *DS / DRM service* experiment, the twelve state changes resulted in only eleven calls to the DRM service. The DS service did not need to call the DRM service when the first task entered the system because that task was schedulable. In the *DS / DRM service* experiment, the average MRT for a new task entering the system in which it could be scheduled was 0.0594 seconds. If the system was not schedulable, meaning the DS service had to call the DRM service, the average MRT was 0.18459 seconds. Tasks leaving the system gave an average MRT of 0.13255 seconds, which is the time it took the DS service to notify the DRM service of potential overload, and the DRM service to recalculate maximum Service Levels for the remaining tasks.

## 5. Related Work

Several other real-time middleware systems provide distributed scheduling similar to that provided by the DS service in this architecture. The Real-Time CORBA 1.0 static scheduling service [21] in TAO provides a distributed scheduling framework for statically scheduled systems. That is, when the requirements of all distributed tasks are known *a priori*, this service provides guaranteed real-time scheduling. It is not able, however, to respond to dynamic changes in requirements, as the DS service does.

The Tempus middleware framework [18] works in dynamic environments, and supports distributable threads. Tempus requires that application timing constraints be specified as time/utility functions. It then builds a distributable thread abstraction on top of POSIX threads. The current implementation of the DS service schedules distributable threads through the RTC 1.2 API using fixed priority local enforcement. However, the pluggable design of the DS service would also allow utility function-based scheduling to be used in place of priority-based scheduling.

The QuO framework [16] also provides resource management capabilities. However, QuO does not address integration of scheduling and resource management at the granularity described in this paper. Rather, QuO offers a higher-level resource management framework within which the services developed in this research can be coordinated with other resource management capabilities.

The DS service in our architecture uses a RTC 1.2 framework [25] to enforce local endsystem scheduling policies. However, the DS service goes above and beyond the requirements of RTC 1.2. The RTC 1.2 specification [25] states that the current standard handles end-to-end

scheduling by scheduling each node independently. Our DS service can support globally-sound scheduling (specified as future work by the RTC 1.2 standard document) where possible, by analyzing entire end-to-end chains and scheduling individual nodes accordingly.

The majority of real-time computing research has focused on the scheduling, analysis and resource allocation for real-time systems whose timing properties and execution behavior are known *a priori*. However, open DRE systems must execute in highly dynamic environments, thereby often precluding accurate characterization of the applications' properties by static models. In such contexts, temporal and execution characteristics can only be determined accurately by empirical observation.

Our approach addresses these issues and removes key limitations. End-to-end real-time performance requirements drive resource allocation decisions, and *a posteriori* resource demands are used in allocation analysis. Our effort differs from [19], which performed end-to-end allocation, but assumed average case workloads for servers; in contrast, we use monitors that provide more accurate resource usage profiles. More importantly, our integrated approach enables fine-grain collaboration between the scheduling and resource management services so that the schedulers and resource manager do not make decisions that may counteract or undermine each other's actions.

## 6. Conclusions

Our research on distributed scheduling, resource management, and endsystem scheduling, once fully integrated, has the potential to overcome many of the key shortcomings of previous layered QoS management solutions. We have demonstrated performance improvements of a simulated DRE application controlled by the integrated distributed scheduling and resource management services through controlled experiments. Our experiment results show that the integrated services improved application performance beyond what was achieved by using the resource management service alone.

We plan further refinements to the current DS / DRM service integration. For example, the total management response time of the joint DS / DRM service management may be reduced by increasing parallelism. For example, making the *rm\_reactive()* method a reliable one-way call would let the DS and DRM services work concurrently. This improvement may be particularly significant when the DS and DRM services run on different host computers.

The group scheduling approach also shows potential advantages for application-customized QoS management, when integrated with distributed scheduling and resource management. The integration of endsystem group scheduling with the integrated DS and DRM services is the next stage of our research effort.

## References

- [1] Tejasvi Aswathanarayana, Venkita Subramonian, Deepti Mokkapti, Hariharn Subramanian, Douglas Niehaus, and Christopher Gill, "Design and Performance of Configurable Endsystem Scheduling Mechanisms," the 11<sup>th</sup> IEEE Real-time Technology and Application Symposium (RTAS), San Francisco, CA, March, 2005.
- [2] Andy Bavier and Larry Peterson, "BERT: A Scheduler for Best Effort and Real-time Tasks," Princeton University Technical Report TR-602-99, March 1999.
- [3] R. Braden, et al., "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification," Network Working Group RFC 2205, September 1997.
- [4] Kevin Bryan, Lisa C. DiPippo, Victor Fay-Wolfe, David T. Fleeman, Christopher D. Gill, David W. Juedes, Chang Liu, Matthew Murphy, Douglas Niehaus, Venkita Subramonian, Lonnie R. Welch, and Jiangyin Zhang, "Towards Integrated CORBA Scheduling and Resource Management Services for Distributed Real-Time and Embedded Systems," the 4<sup>th</sup> Annual Workshop on TAO and CIAO, Arlington, VA, July 16, 2004.
- [5] Matthew Delaney, Lonnie Welch, David Juedes, Chang Liu, and David Fleeman, "RMBench: CORBA Services for Evaluation and Benchmarking of Resource Management Middleware," Presented at the 2004 OMG Workshop of Real-Time and Embedded Systems (RTES), Reston, VA, July 12-15, 2004.
- [6] David Fleeman, Matthew Gillen, Andrew Lenharth, Matthew Delaney, Lonnie Welch, David Juedes, and Chang Liu, "Quality-based Adaptive Resource Management Architecture (QARMA): A CORBA Resource Management Service," the 12<sup>th</sup> International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2004) at IPDPS 2004, Santa Fe, New Mexico, April 26-27, 2004.
- [7] Michael Frisbie, "A Unified Scheduling Model for Precise Computation Control," Master's Thesis, University of Kansas, June 2004.
- [8] Michael Frisbie, Douglas Niehaus, Venkita Subramonian, and Christopher Gill, "Group Scheduling in Systems Software," the 12<sup>th</sup> International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2004) at IPDPS 2004, Santa Fe, New Mexico, April 26-27, 2004.
- [9] Christopher D. Gill, *Flexible Scheduling in Middleware for Distributed Rate-Based Real-Time Applications*, Ph. D. Dissertation, Department of Computer Science, Washington University, May 2002.
- [10] Christopher Gill, Lisa DiPippo, Victor Fay-Wolfe, Douglas Niehaus, and Lonnie Welch, "Mapping a Multi-Level Scheduling Pattern Language to Distributed Real-Time Embedded Applications," Proceedings of the Workshop on Patterns in Distributed Real-Time and Embedded Systems, November 2002.

- [11] Christopher D. Gill, Jeanna M. Gossett, David Corman, Joseph P. Loyall, Richard E. Schantz, Michael Atighetchi, and Douglas C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," the 10<sup>th</sup> IEEE Real-time Technology and Application Symposium (RTAS), Toronto, Canada, May 2004.
- [12] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems: the International Journal of Time-Critical Computing Systems*, special issue on Real-Time Middleware, guest editor Wei Zhao, Vol. 20 No. 2, March 2001.
- [13] Christopher Gill, Douglas Niehaus, Venkita Subramonian, Lisa DiPippo, and Victor Fay-Wolfe, "Resource Rationalizer: A Pattern Language for Multi-Scale Scheduling," Proceedings of the 9<sup>th</sup> Conference on Pattern Language of Programs, Monticello, IL, September 2002.
- [14] Christopher D. Gill, Douglas C. Schmidt, and Ron Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings Special Issue on Modeling and Design of Embedded Systems*, Vol. 91, No. 1, January 2003.
- [15] Pawan Goyal, Xingang Guo, and Harrick M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," the 2<sup>nd</sup> Symposium on Operating Systems Design and Implementation, USENIX, October 1996.
- [16] David A. Karr, Craig Rodrigues, Joseph Loyall, Richard E. Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt, "Application of the QoS Quality-of-Service Framework to a Distributed Video Application," In Proceedings of the International Symposium on Distributed Objects and Applications, Rome, Italy, September 18-20, 2001.
- [17] Yamuna Krishnamurthy, Christopher Gill, Douglas C. Schmidt, Irfan Pyarali, Louis Mgeta, Yuanfang Zhang, and Stephen Torri, "The Design and Implementation of Real-time CORBA 2.0: Dynamic Scheduling in TAO," the 10<sup>th</sup> IEEE Real-time Technology and Application Symposium (RTAS), Toronto, Canada, May 2004.
- [18] Peng Li, Binoy Ravindran, Hyeonjoong Cho, and E. Douglas Jensen, "Scheduling Distributable Real-Time Threads in Middleware," In Proceedings of the International Conference on Parallel and Distributed Computing, 2004.
- [19] P. M. Melliar-Smith, L. E. Moser, V. Kolegaraki, and P. Narasimhan, "The Realize Middleware for Replication and Resource Management," in Proceedings of Middleware '98, pp. 123-138, September 1998.
- [20] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System," the 1<sup>st</sup> Symposium on Operating Systems Design and Implementation, USENIX Association, October 1996.
- [21] Matthew Murphy and Kevin Bryan, "CORBA 1.0 Compliant Static Scheduling Service for Periodic Tasks," Technical Report TR04-297, University of Rhode Island, January 2004.
- [22] Douglas Niehaus, "Improving Support for Multimedia System Experimentation and Deployment," In Workshop on Parallel and Distributed Real-Time Systems, San Juan, Puerto Rico, April 1999. Also appears in Springer Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, ISBN 3-540-65831-9, pp 454-465.
- [23] Douglas Niehaus, et al., "Kansas University Real-Time (KURT) Linux," <http://www.ittc.ukans.edu/kurt/>.
- [24] Object Management Group (OMG), *The Common Object Request Broker: Architecture and Specification*, Revision 2.6, December 2001.
- [25] Object Management Group (OMG), *Real-Time CORBA Specification*, Version 2.0, formal/03-11-01, November 2003.
- [26] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron, "Achieving End-to-End Predictability of the TAO Real-time CORBA ORB," the 8<sup>th</sup> IEEE Real-Time Technology and Applications Symposium (RTAS), San Jose, CA, September 2002.
- [27] Binoy Ravindran, Lonnie R. Welch, and Behrooz A. Shirazi, "Resource Management Middleware for Dynamic, Dependable Real-Time Systems," *The Journal of Real-Time Systems*, Vol. 20, pp. 183-196, Kluwer Academic Press, 2000.
- [28] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau, "Evolving Real-Time Systems Using Hierarchical Scheduling and Concurrency Analysis," In Proceedings of the 24<sup>th</sup> IEEE Real-Time Systems Symposium, Cancun, Mexico, December 2003.
- [29] John Regehr and John A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," In the 22<sup>nd</sup> IEEE Real-Time Systems Symposium, London, UK, December 2001.
- [30] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294-324, April 1998.
- [31] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari and Douglas Niehaus, "A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software," the 4<sup>th</sup> IEEE Real-Time Technology and Applications Symposium (RTAS), Denver, CO, June 1998.
- [32] Jun Sun, *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.
- [33] TimeSys, "TimeSys Linux/RT 3.0," <http://www.timesys.com>.
- [34] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," Proceedings of the 5<sup>th</sup> Symposium on Operating Systems Design and Implementation (OSDI 2002), pp. 255-270, Boston, MA, December 2002.
- [35] Z. Zhu, K. Rajasekar, E. Riseman, and A. Hanson, "Panoramic Virtual Stereo Vision of Cooperative Mobile Robots for Localizing 3D Moving Objects," IEEE Workshop on Omnidirectional Vision (OMNIVIS), 2000.