Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems *

Yuanfang Zhang, Chenyang Lu, and Christopher Gill
Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{yfzhang, lu, cdgill}@cse.wustl.edu

Patrick Lardieri and Gautam Thaker
Advanced Technology Laboratories
Lockheed Martin, Cherry Hill, NJ, USA
{plardier, gthaker}@atl.lmco.com

Abstract

Many mission-critical distributed real-time applications must handle aperiodic tasks with end-to-end deadlines. However, existing middleware (e.g., RT-CORBA) lacks schedulability analysis and run-time enforcement mechanisms needed to give online real-time guarantees for aperiodic tasks. The primary contribution of this work is the design, implementation, and performance evaluation of the first realization of deferrable server and admission control mechanisms for aperiodic tasks in middleware. Empirical results on a KURT-Linux testbed demonstrate the efficiency and effectiveness of our deferrable server and admission control mechanisms in TAO's federated event service.

1 Introduction

Many distributed real-time systems must handle a mix of periodic and aperiodic tasks. Some aperiodic tasks have end-to-end deadlines whose assurance is critical to the correct behavior of the system. For example, in an industrial plant monitoring system, an aperiodic alert event may be generated when a series of periodic sensor readings meets certain hazard detection criteria. This event must be processed on multiple processors within an end-to-end deadline. User inputs and sensor readings may trigger various other real-time aperiodic tasks. A key challenge in such systems is providing online real-time guarantees to critical aperiodic tasks that arrive dynamically.

State of the Art. Schedulability analysis is essential for achieving predictable real-time properties. Aperiodic scheduling has been studied extensively in real-time scheduling theory. Earlier work on aperiodic servers has integrated scheduling of aperiodic and periodic tasks [24, 19, 11, 21, 16, 17, 12, 5, 22], and new schedulability tests based on aperiodic utilization bounds [2] and a new admission control approach [3] were introduced recently. However, despite significant theoretical results on aperiodic scheduling, these results have not been applied to the standards-based middleware that is increasingly being used for developing distributed real-time applications. For example, current implementations of Real-Time CORBA (RT-CORBA) [14] do not provide any of the schedulability tests or run-time mechanisms required by aperiodic servers. As a result, those middleware implementations are currently unsuitable for applications with real-time aperiodic tasks. Admission control has been proposed as an effective approach to handle dynamic real-time tasks in distributed operating systems [23, 20]. However, those kernel-level mechanisms cannot be ported to distributed middleware that lacks fine-grained resource control. Admission control also has been implemented in real-time middleware services [1], but they do not support aperiodic end-to-end tasks, which are essential to many distributed real-time applications.

Research Contributions. To address the limitations of current generation real-time middleware in supporting aperiodic tasks in dynamic distributed real-time systems, we have developed an integrated middleware architecture for end-to-end aperiodic and periodic task scheduling in The ACE ORB (TAO) [9]. We have developed what are to our knowledge the first middleware-layer (1) mechanisms for deferrable servers, in TAO's federated event service [8]; and (2) admission control service supporting both aperiodic and periodic end-to-end tasks. Our work bridges an important gap between aperiodic scheduling theory and state-of-the-art real-time middleware. Applying that theory to a real-world middleware environment faces several important challenges as it requires (1) consideration of the specific requirements of the system model in applying that theory, and (2) highly efficient design and implementation of the run-time mechanisms on standard operating system platforms. Our empirical results on a Linux testbed demonstrate the success of our approach in supporting deferrable servers and online admission control for both aperiodic and peri-

*This work was supported in part by the DARPA Adaptive and Reflective Middleware Systems (ARMS) program (contract NBCHC030140) and NSF CAREER award CNS-0448554.

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

odic end-to-end tasks, efficiently in middleware.

Section 2 introduces the specific system model addressed by this research, and provides background information on aperiodic scheduling approaches. Section 3 presents our middleware architecture for supporting aperiodic task scheduling end-to-end in distributed real-time systems. Section 4 evaluates the performance of our approach. Finally, we offer concluding remarks in Section 5.

2 Background

In this section we describe a representative distributed real-time application that has motivated the work presented in this paper, and consider alternatives for scheduling the aperiodic tasks that feature prominently in that application.

2.1 Shipboard Computing System Model

Military shipboard computing [26] is moving toward a common computing and networking infrastructure that hosts the mission execution, mission support and quality of life systems required for shipboard operations. The objective of the middleware architecture presented in this paper is to ensure that in the complex large-scale distributed real-time computing environments envisioned, mission critical and safety critical tasks will meet their real-time performance requirements, even in the presence of a myriad of other competing non-critical real-time tasks.

The physical system architecture consists of a number of display consoles that host primarily human/computer interface software, and are connected by a real-time network to a large number of inter-connected servers that host the software that delivers much of the system's mission computing capability. The software running on this distributed infrastructure comprises a mix of periodic and aperiodic tasks, which are subject to a mix of critical and non-critical performance requirements.

On the servers, many tactical applications are implemented as end-to-end tasks each consisting of multiple subtasks that may be located on different processors. An example of such an application is sensor data processing, which consists of multiple subtasks. The majority of this data processing is periodic and non-critical. However, if a series of sensor reports meets certain threat criteria, an urgent self defense mode may be enabled. Further processing of the data becomes a critical aperiodic task with a deadline, i.e., to make an engagement decision. Should the decision be to engage, a critical periodic task is then launched, i.e., to manage countermeasures.

The shipboard computing system model is representative of many complex distributed real-time systems. Such a system is composed of a set of periodic and aperiodic tasks. A task is composed of a chain or a graph of subtasks which may be located on multiple processors. A subtask cannot be released until its predecessor has been completed. A task is subject to an end-to-end deadline which may be critical or non-critical. Since new aperiodic and periodic tasks may arrive dynamically, it is impossible to provide realistic offline guarantees of the schedulability of the system. Instead, an admission control strategy is needed to provide on-line guarantees to aperiodic tasks. Our middleware architecture is designed to support this general system model, though some of the specific policies supported by our current implementation are driven by the characteristics of the shipboard computing system model in particular.

2.2 Aperiodic Scheduling Approaches

We now give a brief review of two aperiodic scheduling techniques that can be applied to the system model described in Section 2.1. For each approach we describe both the required run-time scheduling mechanisms and the schedulability analysis. Although each approach may have various possible policies, in the following we only describe the particular set of policies provided by our middleware.

Aperiodic Utilization Bound. According to the aperiodic utilization bound (AUB) analysis [2], a system achieves its highest schedulable synthetic utilization bound under the End-to-end Deadline Monotonic Scheduling (EDMS) algorithm. Under EDMS, a subtask has a higher priority if it belongs to a task with a shorter end-to-end deadline. The subtasks of a given task are synchronized by a greedy protocol, because the AUB analysis does not require their inter-release times to be bounded. Note that AUB does not distinguish aperiodic from periodic tasks. All tasks are scheduled using the same scheduling policy. In the AUB analysis, the set of *current* tasks $S(t)$ at any time t is defined as the set of tasks that have released but whose deadlines have not expired. Hence, $S(t) = \{T_i | A_i \leq t < A_i + D_i\}$, where A_i is the release time of the first subtask of task T_i, and D_i is the deadline of task T_i. The synthetic utilization of processor *j*, $U_j(t)$, is defined as the sum of individual subtask utilizations on this processor, accrued over all current tasks. Under EDMS task T_i will meet its deadline if the following condition holds [2]:

$$\sum_{j=1}^{n_i} \frac{U_{V_{ij}}(1 - U_{V_{ij}}/2)}{1 - U_{V_{ij}}} \leq 1 \qquad (1)$$

where V_{ij} is the j^{th} processor that task T_i visits.

To reduce the pessimism of the AUB analysis, a *resetting rule* is introduced in [2]. When a processor becomes idle, the contribution of all completed aperiodic subtasks to the processor's synthetic utilization is removed.

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

Deferrable Servers. We choose deferrable server (DS) [24] instead of more sophisticated servers because it allows a simple and efficient implementation at the middleware layer. In the DS approach, a *server* executes all aperiodic subtasks on a processor. A server has a *budget* and a *period*. The budget is replenished at the beginning of each period. The budget decreases whenever the server is executing an aperiodic subtask, and it is preserved till the end of the current period when the server is idle. A server can execute aperiodic subtasks as long as its budget has not been exhausted.

In our DS implementation, a server executes aperiodic subtasks in the order of their end-to-end deadlines in a non-preemptive fashion. To fairly compare with the alternative approach (AUB), all periodic tasks and the server itself are scheduled by a preemptive end-to-end deadline monotonic scheduling (EDMS) policy. The subtasks of each aperiodic task are synchronized by a greedy protocol. The subtasks of each periodic task are synchronized by the release guard protocol, so that the inter-release times of periodic subtasks are bounded by their periods. A heuristic method is used to select the budgets and periods of deferrable servers. The method is not shown here due to space limitations and can be found in [27].

We adapt the method proposed in [4] to compute the worst-case response times of each aperiodic subtask. An aperiodic task is schedulable if the sum of all its subtasks' worst-case response times is less than the deadline. For periodic tasks, we apply the time demand analysis method to determine whether all periodic tasks remain schedulable in the presence of deferrable servers according to [24, 13]. Since the release guard protocol is used to synchronize periodic subtasks, the worst-case end-to-end response time of a periodic task is the sum of the worst-case response times of all its subtasks on different processors [25].

While the theories for DS and AUB have been investigated extensively in the literature, neither has been realized in distributed middleware systems. A key contribution of our work is to bridge the gap between these theories and their application through the development of an integrated middleware architecture, which is described in Section 3. Furthermore, we also provide the first empirical comparison of the AUB and DS approaches on an actual middleware platform, in Section 4.

3 Middleware Architecture

To support the general system model presented in Section 2.1, we have developed a new middleware architecture based on TAO's federated event service [8]. The key feature of our architecture is an *integrated scheduling framework* for a mix of critical/non-critical and aperiodic/periodic tasks. Our integrated scheduling framework is composed of three key components: (1) end-to-end middleware services for scheduling and dispatching both aperiodic and periodic tasks; (2) deferrable server mechanisms; and (3) an *admission controller* that provides on-line admission control and schedulability tests for tasks that arrive dynamically at run time.

Our integrated end-to-end scheduling framework can be configured to support either the AUB or the DS approach. If the middleware is configured to support the AUB approach, the end-to-end scheduling service schedules all tasks using the EDMS algorithm, and the admission control service uses the AUB analysis to make admission decisions. On the other hand, if the middleware is configured to support the DS approach, the end-to-end scheduling service runs the EDMS algorithm with the DS mechanisms, and the admission control service uses the corresponding DS analysis to make admission decisions.

In this section, we first describe the implementation of end-to-end tasks using TAO's federated event service, and give an overview of the mechanisms used by our integrated end-to-end scheduling service to ensure correct preemption control over aperiodic and periodic tasks. We then offer a detailed discussion of the deferrable server mechanisms used in our middleware architecture. Finally, we describe the design and implementation of our admission control service, which can provide preemption control over critical and non-critical tasks.

3.1 Middleware Services

To host TAO's federated event service, each processor has its own event channel (EC), and the ECs exchange remote events via Gateways, as described in [8]. Subtasks are implemented as event suppliers and consumers. The supplier pushes events which trigger subtask execution in a consumer. Upon the completion of the subtask, the Gateway located on the supplier pushes an event to the processor where the next subtask is located.

A crucial feature of the shipboard computing system model described in Section 2.1 is the presence of both periodic and aperiodic tasks. To provide integrated scheduling of both aperiodic and periodic tasks, careful design of middleware dispatching mechanisms is needed. The Kokyu dispatching framework [7] is used in our system to provide such real-time dispatching of events to functions that execute the corresponding subtask processing.

When the deferrable server strategy is used, Kokyu is configured to use the preemptive EDMS algorithm to schedule periodic tasks. As shown in Figure 1, each periodic event is assigned to a specific dispatching queue according to its end-to-end deadline. Each dispatching queue has a thread whose priority is determined by the end-to-end deadline of the events in its queue based on the EDMS policy.

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

Each dispatching thread removes the event from the head of its queue and runs its subtask function until it completes or is preempted by a higher priority dispatching thread. To enable end-to-end response time analysis for the periodic tasks, we implemented release guards for the event channels [28] as an extension to the Kokyu dispatching framework. The release guard mechanism is enabled only for periodic tasks. A single deferrable server thread is used to dispatch aperiodic events on each processor in our approach, and runs at a higher priority than any of the periodic task dispatching threads, as we discuss in Section 3.2.

When the aperiodic utilization bound strategy is used, Kokyu is configured to use the preemptive EDMS algorithm to schedule *both* aperiodic and periodic tasks. It uses similar dispatching mechanisms to the DS approach, but there is no deferrable server thread and the release guard mechanism is disabled because it is not needed for AUB.

3.2 Deferrable Server Mechanisms

To support aperiodic tasks in middleware, we have integrated efficient support for deferrable servers within TAO's federated event service as is described in Section 3.1. In this section we present the design and implementation of the deferrable server mechanisms we have developed.

Design Challenges and Decisions. To simplify schedulability analysis and to reduce implementation complexity and run-time overhead we chose to use one deferrable server per processor, though in general multiple deferrable servers may be used on one processor to execute aperiodic tasks with different priorities. All aperiodic subtasks on the same processor are executed by this server in non-preemptive EDMS order. It is challenging to implement bandwidth preserving servers efficiently on top of standard operating systems that do not support CPU reservations, because the budget and execution of the server must be managed in user space. We chose to implement our deferrable server mechanisms on ACE [10] and KURT-Linux, a Linux platform representative of standard operating systems that support fixed priority scheduling. An importance advantage of our deferrable server mechanisms is that they are portable to COTS platforms used by many DRE applications.

Solution Approach. Our deferrable server mechanism is implemented by a pair of threads on each processor: a *server* thread that processes aperiodic events and a *budget manager* thread that manages the budget and the execution of the server thread. The server thread, which is assigned the second highest operating system priority, executes all aperiodic subtasks. If an aperiodic event arrives and the server thread has not used up its budget in the current period, the aperiodic subtask will be able to preempt any other running periodic task. The budget manager thread runs at the highest operating system priority to manipulate the consumption and replenishment of the server's budget. Its higher priority allows it to interrupt the server thread when the budget is exhausted in the current period. The two threads share a timer queue and two variables, $left\_exec$ and $left\_budget$. $left\_exec$ keeps track of the remaining execution time for the currently running aperiodic subtask, and $left\_budget$ keeps track of the remaining budget in the current period. Both variables are protected by a mutex.

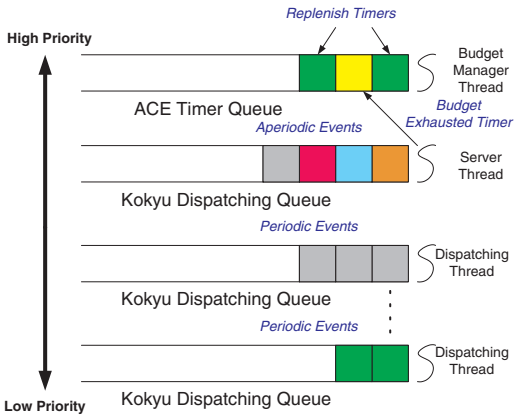


Figure 1. Deferrable Server in TAO

As we show in Figure 1, the server thread dispatches all aperiodic events arriving at the server's Kokyu dispatching queue, to the functions that execute their corresponding subtasks. Before dispatching an aperiodic event, the server thread will first determine if there is enough budget left for the execution of the subtask to complete in the **current** period. Since the actual usable budget, $actual\_budget$, in the current period is the minimum of the remaining budget and the remaining time in the current period, the server thread compares the execution time C of the subtask with $actual\_budget$.

If there is not enough budget left in the current period for the subtask to complete, the server thread will insert a *budget exhausted* timer into a timer queue owned by the budget manager thread. The budget manager thread sits in a `select` system call and waits for the earliest timer to fire. The *budget exhausted* timer will fire when the remaining budget has been used up, and will generate a *budget exhausted* event. This budget exhausted event is handled by the budget manager thread, which runs at the highest priority. When this happens, the budget manager thread sends the SIGSTOP signal to the server thread to suspend it.

The budget manager thread also handles a timer-driven event called *replenish*, which occurs at the beginning of each server period. When this happens, the budget manager thread will replenish the budget B and will also send the SIGCONT signal to the server thread to resume it (if it had been suspended previously). It may set a new *budget exhausted* timer which will fire when the replenished bud-

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

get B has been used up, if it finds the remaining execution time of the current aperiodic subtask is longer than the replenished budget B. We summarize the three operations (*Dispatching*, *Exhausted* and *Replenish*) involved in our deferrable server mechanism in Figure 2. Each operation is partitioned into one or more steps. Each step is represented by a label shown on the right. For example, step S1 includes all statements between "S1 begin" and "S1 end", inclusive.

To reduce the priority inversion relative to the EDMS policy, the server thread is assigned a shorter period than all the periodic tasks on the same processor. The tuning method for the deferrable server is not shown due to space limitations and can be found in [27].

```
Server Thread (before Dispatching an aperiodic event):
    if actual_budget < C then                          S1 begin
        left_budget = 0;
        left_exec = C - actual_budget;
        set_timer = TRUE;
    else
        left_budget = left_budget - C ;
        left_exec = 0;                                 S1 end
    if set_timer = TRUE then                           S2 begin
        set exhausted timer;                           S2 end
-------------------------------------------------------------
Budget Manager Thread (on an Exhausted Event):
    suspend server thread;                             BE1
-------------------------------------------------------------
Budget Manager Thread (on a Replenish Event):
    if B < left_exec then                              BR1 begin
        left_budget = 0;
        left_exec = left_exec - B ;
        set_timer = TRUE;
    else
        left_budget = B - left_exec;
        left_exec = 0;                                 BR1 end
    if server thread is suspended then                 BR2 begin
        resume server thread;                          BR2 end
    if set_timer = TRUE then                           BR3 begin
        set exhausted timer;                           BR3 end
```

Figure 2. Deferrable Server Operations

We note that our budget management mechanism will function properly as long as the server thread runs at a higher priority than the other dispatching threads, and can only be interrupted by the budget manager thread. Our implementation enforces this execution model by assigning appropriate real-time priorities to Linux threads. As our implementation is based on ACE, it is portable to other platforms that support fixed priority scheduling. The portability across different COTS platforms is an inherent advantage of our middleware approach over a kernel-based approach.

We also note that our deferrable server mechanisms are especially suitable for applications that all aperiodic tasks are critical, such as the shipboard computing application described in Section 2.1. The higher priority of the server thread prevents periodic tasks from interfering with aperiodic tasks, while the budget management mechanism of the deferrable server bounds the interference of aperiodic tasks with the response times of periodic tasks.

Synchronization Overhead. A middleware-layer implementation necessarily introduces more overhead than an implementation in the operating system kernel. As an example of such additional overhead we now discuss the overhead related to the timer queue shared by the budget manager and server threads. In our implementation, all threads are synchronized when they access the shared timer queue. Only one owner thread can access the queue at a time and other threads wait in a waiting queue guarded by a mutex. Since the budget manager thread which handles the timer triggered events needs exclusive access more often, it is the owner of the timer queue by default even when it is blocked in the select call.

In our implementation, we use an ACE reactor [18] object to encapsulate the select call in order to balance our solution's portability and efficiency. The ACE reactor not only wraps an event demultiplexing mechanism such as select(), but also provides a special event handler for a notification pipe that can be used to break out of a select call when no other event has arrived. When the server thread tries to insert a timer into the timer queue, it notifies the budget manager thread blocking on the select call to return early via the reactor's notification pipe, and then the server thread sleeps in the waiting queue. The notified budget manager thread first inserts itself into that waiting queue, then releases ownership to the server thread. After the server thread inserts the budget exhausted timer into the timer queue, it releases ownership back to the budget manager thread which goes back to sit in the select system call. When the budget exhausted timer fires, the budget manager thread returns from the select call and suspends the server thread. Therefore, whenever the server thread sets the budget exhausted timer it incurs multiple context switches with the budget manager thread and related synchronization overhead. This is part of the cost of supporting deferrable servers at the middleware layer on top of a standard OS that does not support CPU reservation. Fortunately, our empirical results show that this cost is negligible for all but the most stringent distributed real-time applications (see Section 4.1).

3.3 Admission Control Service

On-line admission control offers significant advantages to the shipboard computing system model by (1) providing online schedulability guarantees to tasks arriving dynamically and (2) (re)allocating resources based on task criticality[1]. Our admission control service adopts a centralized

[1] Although the EDMS scheduling policy does not consider task criticality, our online admission control service can guarantee the preference of critical tasks over non-critical tasks.

architecture, which employs a central admission controller (AC) running on a separate processor from the application processors (APs). The AC and APs communicate through the federated event service. When a new task arrives at an AP, the AP can release the task only if the AC verifies that its execution on multiple APs can meet its end-to-end deadline without violating the schedulability of the system. Applications register every task's scheduling-related information, including its deadline, period (if it is periodic), and its subtasks' locations and execution times, to the AC at system deployment time. In our current implementation, we assume no dependence between different end-to-end tasks. A key advantage of the centralized architecture is that it does not require synchronization among distributed admission controllers. In contrast, in a distributed architecture the ACs on multiple APs may need to coordinate and synchronize with each other in order to make correct decisions, because admitting an end-to-end task may affect the schedulability of other tasks located on multiple APs. A potential disadvantage of the architecture is that the AC may become a communication bottleneck and affect scalability. However, this is not a concern in the shipboard computing system model, in which processors are connected by high-speed real-time networks. Furthermore, the computation times of both schedulability analyses are significantly lower than task execution times in typical shipboard computing applications. As our empirical results in Section 4.2 show, the AC overheads are acceptable and are dominated by the communication delays.

Admission Control Process. Our integrated middleware architecture may be configured to support either the AUB or DS approach. In the following, we describe the interactions between the AC and the APs assuming the middleware is configured to support the AUB approach. The admission control process with the DS approach is similar except that the admission controller uses a different schedulability analysis (as is described in Section 2.2).

Figure 3 shows a detailed view of the interactions between the AC and APs, and of the mechanisms that support them. When a task arrives at its first AP, the AP does not release the task immediately. Instead, it holds the task in a waiting queue (step 1) and sends an "Accept?" event to the AC to notify it (step 2). The AP also sets a timer that will fire when the laxity of the task reaches zero. When the "Accept?" event is delivered to the AC by the federated event channel (step 3), the AC performs schedulability analysis (step 4). If all admitted task can still meet their deadlines if the new task is admitted, the AC sends back a "Release" event to the AP (steps 5 and 6) and sets a timer to fire at the deadline of new admitted aperiodic task (step 7). Upon receiving the "Release" event, the AP dequeues the waiting task and releases it (step 8).

If the system would be unschedulable if the new task was admitted, the AC holds its request in a waiting queue (step 5"). All tasks in the waiting queue are ordered by their increasing laxity. When the number of active tasks in the system is decreased – at the admitted aperiodic task deadlines (step 9) or when any processor becomes idle (steps 10 and 11) – the AC analyzes schedulability and may release waiting tasks with lower laxity if the system becomes schedulable. The AP rejects a task by removing it from the waiting queue when its laxity reaches zero (step 12).

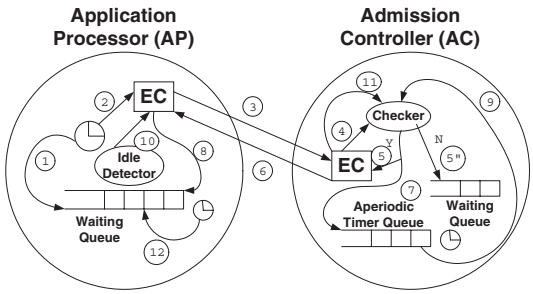


Figure 3. Admission Control Mechanisms

Criticality-awareness Policy. It is important to consider criticality in admission control in the shipboard computing system model, which includes both critical and non-critical tasks. When the AC finds the system would be unschedulable if it accepted a new critical task, the admission controller may eject non-critical periodic tasks previously admitted into the system. In our current implementation, the middleware does not stop the current job of the non-critical periodic task to be ejected if it has already been released. This is because terminating a running job in a distributed middleware setting may incur significant overhead and risk of error. Instead, our AC employs a light-weight ejection policy. The AC sends an "Eject" event to the AP hosting the first subtask of the task to be ejected. When the ejected periodic task tries to release its next job, it is required to go through the admission control process again. There are many possible policies for selecting the tasks to be ejected from the system. Our AC currently randomly chooses the victims from all accepted non-critical periodic tasks. While our current admission control policy only considers two classes of tasks (critical vs. non-critical), it may also be extended to support more sophisticated optimization techniques to maximize total system utility (e.g., [1, 15]).

Resetting Rule. As is mentioned in Section 2.2, the resetting rule may reduce the pessimism of the AUB schedulability test significantly. We have implemented the resetting mechanism in our middleware as follows. Each AP records completed jobs of its aperiodic subtasks. Whenever the processor is idle, a lowest priority thread called an *idle detector*

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

begins to run. Its main job is to report the completed aperiodic jobs to the AC through an "Idle" event. To avoid reporting repeatedly, the idle detector only reports when there is a new completed aperiodic job whose deadline has not expired. Note that the resetting rule is only applied to aperiodic tasks, because (to avoid significant overhead) the AC does not consider individual jobs of periodic tasks in the schedulability analysis.

Delay Accounting. Three types of delays should be considered in the admission tests for both AUB and DS. The first type of delay is the round trip delay between an AP and the AC, which includes delays in both the middleware and the network. The second type of delay is the time a task stays in the waiting queue on the AC. Note that we do not consider the waiting time as part of the round trip delay, because it is caused by system overload instead of by the AC mechanism itself. Finally, the third type of delay is the event communication delay between subtasks located on different APs. Our admission test accounts for these delays by deducting them from the end-to-end deadline when assessing the schedulability of a task.

4 Empirical Evaluation

This section presents experiments we conducted to evaluate the efficiency and effectiveness of our middleware architecture. The experiments described in this section were performed using ACE/TAO version 5.1.4/1.1.4 on a testbed consisting of four machines connected by a 100Mbps Ethernet switch. Two of them are Pentium-IV 2.5GHz machines, and the other two are Pentium-IV 2.8GHz machines. Each of them has 500MB RAM and 512KB cache, and runs version 2.4.22 of the KURT-Linux operating system [6]. These platforms provide a CPU-supported timestamp counter with nanosecond resolution.

The data stream user interface (DSUI) and data stream kernel interface (DSKI) frameworks are provided with the KURT-Linux distribution. The DSUI is used to record information from the middleware and application layers, while the DSKI is used to collect information like the frequency of thread context switches at the kernel level. By using both DSUI and DSKI, we obtain a precise accounting of operation start and stop times, thread context switches, and other relevant events across multiple system levels.

To account for the communication delay in the schedulability analysis on our experimental platform, we pushed an event back and forth between two machines 1000 times, measured each round-trip time on one machine, and then divided the maximum round trip time by 2 to obtain the *approximate* maximum communication delay, which was 553 μs. In practice DRE system environments such as those for shipboard computing usually use real-time networks that can provide guarantees on worst-case communication delays. When our middleware is integrated with such real-time networks, the admission controller also can account for the delay bounds guaranteed by the networks.

4.1 Deferrable Server Overhead

To measure the run-time overhead incurred by our deferrable server mechanisms, we first randomly generated a task set consisting of 9 tasks including 4 critical aperiodic tasks and 5 critical periodic tasks. The number of subtasks per task is uniformly distributed between 1 and 3. Subtasks are randomly assigned to 3 processors. Task deadlines are randomly chosen between 250 ms and 10 s. The periods of periodic tasks are equal to their deadlines. The arrival of aperiodic tasks follows a Poisson distribution. The synthetic utilization of every processor is 0.5, if all tasks arrive simultaneously. After running the task set for 5 minutes, we extracted the data recorded by DSUI and DSKI during that interval, and then calculated the overhead.

BR	BR1	BR2	BR3	Total
	8.269/ 37.966	7.172/ 8.482	10.408/ 17.140	15.556/ 66.835
BE	BE1(Total)			
	8.863/21.832			
Server	S1	S2	Sync	Total
	21.501/ 31.243	43.212/ 45.532	62.063/ 71.747	78.799/ 157.926

Table 1. Mean/Max Overhead of Deferrable Server Mechanisms (μs)

Budget Manager Thread. In each period of the deferrable server, the budget manager thread is released at most twice. It is released at the beginning of each period to handle the *budget replenish* (BR) event, and it may be released again to suspend the execution of the server thread when the *budget exhausted* (BE) event happens. Since not every step of the deferrable server mechanism (shown in Figure 2) is executed for each release, we measure the mean and maximum execution time of each step during a 5 minute run, as is shown in Table 1[2]. Since in each DS period, the replenish operation runs once and the exhausted operation runs at most once, the maximum budget manager overhead per DS period is approximately the sum of the maximum overheads of its replenish and budget exhausted operations. Our measurements showed that the maximum overhead of the budget manager thread is 88.667 μs. The mean overhead is significantly lower, because some steps may not be invoked for some releases. For example, BR2 is executed

[2]The slight difference between the total overhead of BE and the sum of all 3 steps is consistent with the cost of DSUI and DSKI instrumentation.

only if the server thread has been suspended because of the *budget exhausted* event. Step BR3 is executed only if the remaining execution time of the current aperiodic task is longer than the replenished budget. For the replenish operation, its mean overhead for each DS period is about 15.556 μs, which is close to the mean overhead of its first step, BR1. The mean and maximum overhead of BR1 show a significant difference due to the variable overhead for entering and exiting a mutex provided by Linux.

Server Thread. The server thread incurs overhead every time it dispatches a subtask. The maximum overhead of the server thread is 157.926 μs per subtask. As discussed in Section 3.2, synchronization between the budget and server threads incurs overhead when the server thread inserts a budget exhausted timer, which can be as high as 71.747 μs. The S2+Sync overhead when the server thread inserts an exhausted timer causes the difference between the mean and maximum overhead of the server thread. If no exhausted timer is needed, the total overhead of the dispatching operation (22.449 μs) is close to the overhead of its first step (S1). In summary, our deferrable server mechanisms incur a maximum overhead of 88.667 μs per DS period and 157.926 μs per aperiodic subtask, which are acceptable for many distributed real-time system environments, such as those for shipboard computing.

4.2 Admission Control Overhead

To evaluate the overhead of our admission control service, we used 3 processors to run applications and another processor to run the admission controller. The workload is randomly generated in the same way as described in Section 4.1, except that the synthetic utilization of every processor is 0.4 if all tasks arrive simultaneously. The experiment ran for 5 minutes.

Round Trip Delay. Before releasing an aperiodic job or the first job of a periodic task, the application processor will wait for the "Release" event from the admission controller. The time interval from when one job arrives until the processor actually releases it, is called the round trip delay. The round trip delay includes delays at both the network layer and the middleware layer. The admission control service must achieve short round trip delays because they contribute to the response time of a task. The mean/max round trip delays of the admission control services with AUB and DS are 1072/1356 μs and 1083/1315 μs respectively[3]. The results only consider the cases when a task is admitted by the admission controller when it arrives to separate the overhead introduced by admission control from the wait time caused by system overload. Interestingly, the difference between the round trip delays with AUB and DS is negligible.

To analyze the round trip delay in detail, we also measured the delay introduced by each step in the admission control process (shown in Figure 3), when the AUB approach is used. From Table 2, we can see that the communication delays between the application processors (AP) and the admission controller (AC) dominate the other admission control steps. The communication delay between an AP and the AC is different from the communication delay between APs because the events have different payload sizes. This accounting explains the similar round trip delays regardless of the admission test used. Overall, our empirical results show that our admission control service only introduces moderate delay penalties that are acceptable for shipboard computing and other DRE system environments.

host	step	description	mean	max
AP	1	Queue new job to wait	32	41
	2	Send "Accept?" event	106	127
both	3,6	Communication delay	355	435
AC	4	Schedulability analysis	23	31
	5	Send "Release" event	85	102
	5"	Queue request to wait	8	9
	7	Set aperiodic timer	78	85
AP	8	Release new job	75	92
AC	9	Timer fires, update util	41	107
AP	10	Send idle event	143	151
AC	11	Rcv idle event, update util	22	27
AP	12	Timer fires, reject job	52	61

Table 2. Admission Control Step Delays (μs)

4.3 Acceptance Ratios

We perform two sets of experiments to compare different scheduling approaches. The performance metric is the *acceptance ratio*, i.e., the fraction of jobs accepted by the admission controller. The first set of experiments compares the acceptance ratio when different schedulability analyses and their corresponding scheduling algorithms are used by the admission controller. The second set of experiments evaluates the effectiveness of the criticality-aware admission policy. The workload characteristics used in our experiments are representative of the applications running in shipboard computing environments.

Comparison of Schedulability Analyses. We compare three methods for performing schedulability tests: AUB, AUB without reset, and DS, when they are used in our admission control service. The resetting mechanism is disabled when AUB without reset is used. The comparison of AUB to AUB without reset evaluates the effectiveness of the resetting rule in reducing the pessimism of the AUB

[3]For clarity we present the results at the granularity of microsecond, although our measurement using DSUI/DSKI has nanosecond granularity.

test. While the effect of the resetting rule has been studied through simulations [2] previously, we provide its first implementation and empirical evaluation on a real middleware platform. Our results are also the first empirical comparison of the AUB and DS mechanisms.

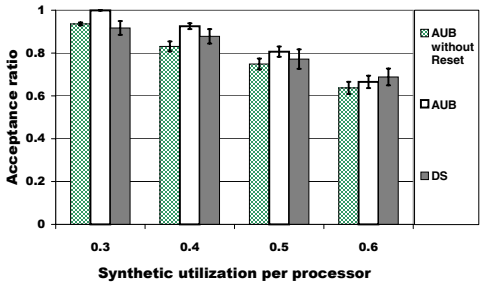


Figure 5. Acceptance Ratio Comparison

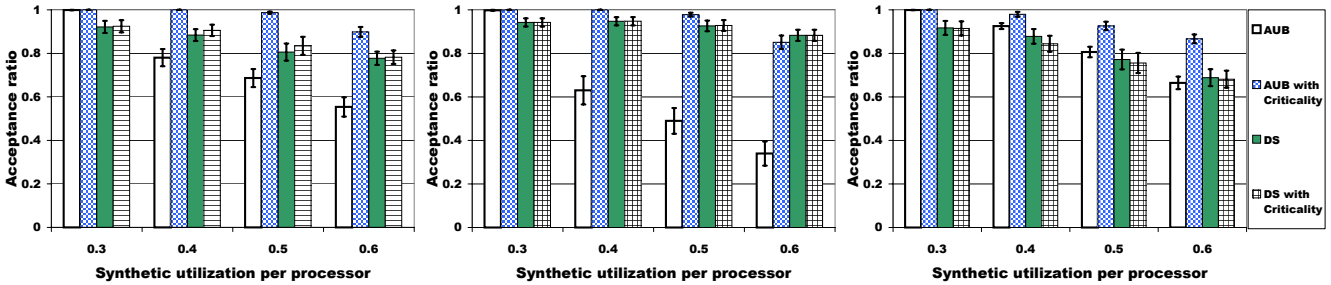
We changed the total synthetic utilization of all subtasks on each processor systematically in different experiments, and randomly generated 60 task sets for each total synthetic utilization per processor. Note that the total synthetic utilization is calculated assuming there is a current instance of each (aperiodic or periodic) task on a processor. It is different from the *instantaneous* synthetic utilization on the processor at run time. Each task set includes 4 aperiodic tasks and 5 periodic tasks with similar characteristics to the workloads used in earlier experiments. All tasks were made equally critical in this set of experiments.

Figure 5 shows the mean and 90% confidence interval of the acceptance ratio for different synthetic utilizations. The resetting rule consistently increases the acceptance ratio under AUB. For instance, when the synthetic utilization is 0.4, AUB accepts 92.5% of the jobs, 9.4% higher than AUB without the resetting rule. The benefit of resetting decreases at higher load. This is expected because the processor rarely idles under high load. We also observe that the acceptance ratios of DS and AUB remain close except under light load, when AUB outperforms DS.

Criticality-aware Admission Control. To evaluate the effectiveness of the criticality-aware admission policy, we compare four different admission policies: AUB, AUB with criticality, DS, and DS with criticality. With AUB and DS the admission controller does not eject non-critical tasks to accommodate new critical tasks. To "stress test" the admission control policies we randomly generated 60 task sets for each of four total synthetic utilizations: 0.3, 0.4, 0.5 and 0.6. Each task set includes 9 tasks. 4 of them are aperiodic tasks. To be consistent with the shipboard computing system model described in Section 2.1, all aperiodic tasks are critical. 2 of the periodic tasks are critical, and the other 3 are noncritical.

We first study the effect of criticality on AUB. As shown in Figure 4(a), the acceptance ratio for critical tasks is improved significantly when criticality is considered by the AUB approach. For instance, when the synthetic utilization is 0.4, AUB with criticality accepts all critical jobs, while AUB only accepts 78% of the critical jobs. As shown in Figure 4(b) considering criticality has an especially significant effect on aperiodic tasks. With the synthetic utilization of 0.4, AUB with criticality accepts all aperiodic jobs (all of which are critical), while AUB only accepts 63% of the aperiodic jobs. The significant benefit to aperiodic tasks is attributed to the fact that AUB with criticality can eject non-critical periodic tasks to accommodate critical aperiodic tasks. As our middleware reserves utilization for periodic tasks to avoid the overhead of making admission decision on every periodic job, the admission control service is more pessimistic for periodic tasks than for aperiodic tasks. Therefore, by replacing non-cirtical tasks with critical aperiodic tasks, AUB with criticality not only improves the acceptance ratio for critical and aperiodic tasks, but also improves the overall acceptance ratio for all tasks, as is shown in Figure 4(c).

On the other hand, considering criticality in DS only achieves moderate improvement in the acceptance ratio of critical tasks, and has little effect on the acceptance ratio of aperiodic tasks or the overall acceptance ratio of all tasks. This is because in DS all aperiodic tasks have their predetermined budgets on all processors which are not affected by admitting or ejecting any periodic tasks. As a result with



(a) Acceptance Ratio for Critical Tasks (b) Acceptance Ratio for Aperiodic Tasks (c) Acceptance Ratio for All Tasks

Figure 4. Effect of Criticality-aware Admission Control

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE

the DS approach the admission controller can no longer accommodate aperiodic tasks by ejecting non-critical periodic tasks. The improvement in the acceptance ratio of critical tasks is only due to the cases when a non-critical periodic task is ejected to accommodate critical periodic tasks, while the acceptance ratio of non-critical tasks decreases accordingly. The figure for the acceptance ratio of non-critical tasks is not shown due to space limitations.

In summary, our empirical results have shown that our implementation of the resetting mechanisms can effectively reduce the pessimism of the AUB-based admission control in an actual middleware system. Furthermore, AUB can be more effective than DS for critical and aperiodic tasks when combined with a criticality-aware admission policy. This feature makes AUB with criticality a particularly attractive approach for the shipboard computing system model with a mix of critical aperiodic tasks and critical/non-critical periodic tasks.

5 Conclusions

Our work represents a promising step toward developing integrated end-to-end scheduling services for aperiodic and periodic tasks in distributed real-time middleware. We have designed and implemented (1) novel deferrable server mechanisms (integrated with TAO's federated event service), and (2) an efficient middleware service that provides online admission control and schedulability guarantees. Empirical results showed that (1) our deferrable server mechanisms are highly efficient on a Linux platform and (2) the combination of a criticality-aware admission policy and the aperiodic utilization bound strategy is especially effective for representative workloads with a mix of critical aperiodic tasks and critical/non-critical periodic tasks.

References

[1] T. Abdelzaher, E. Atkins, and S. K. QoS Negotiation in Real-Time Systems and its Application to Automated Flight Con trol. *IEEE Transactions on Computers*, Nov. 2000.

[2] T. F. Abdelzaher, G. Thaker, and P. Lardieri. A Feasible Region for Meeting Aperiodic End-to-end Deadlines in Resource Pipelines. In *ICDCS*, 2004.

[3] B. Andersson and C. Ekelin. Exact Admission-Control for Integrated Aperiodic and Periodic Tasks. In *RTAS*, 2005.

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Springer-Verlag New York Inc, 2004.

[5] R. I. Davis, K. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *RTSS*, 1993.

[6] Douglas Niehaus, *et al.*. Kansas University Real-Time (KURT) Linux. `www.ittc.ukans.edu/kurt/`, 2004.

[7] C. Gill, D. C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), Jan. 2003.

[8] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA*, 1997.

[9] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[10] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/, Vanderbilt University.

[11] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *RTSS*, 1987.

[12] J. P. Lehoczky and S. R. Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *RTSS*, 1992.

[13] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[14] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, Aug. 2002.

[15] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *RTSS*, 1997.

[16] S. Ramos-Thuel and J. P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-prioriry systems. In *RTSS*, 1993.

[17] S. Ramos-Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed priority systems using slack stealing. In *RTSS*, 1994.

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

[19] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *RTSS*, 1986.

[20] S. Sommer and J. Potter. Operating System Extensions for Dynamic Real-Time Applications. In *RTSS*, 1996.

[21] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1(1):27–60, 1989.

[22] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems, Real-Time Systems. *The Journal of Real-Time Systems*, 10(2), 1996.

[23] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Systems. *IEEE Software*, May 1991.

[24] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[25] J. Sun. *Fixed priority scheduling of end-to-end periodic tasks*. PhD thesis, UIUC, 1997.

[26] United States Navy. Total Ship Computing Environment (TSCE). peoships.crane.navy.mil/ddx/tsce.htm.

[27] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware. Technical Report WUCSE-2005-57, WUSTL, 2005.

[28] Y. Zhang, B. Thrall, S. Torri, C. Gill, and C. Lu. A Real-Time Performance Comparison of Distributable Threads and Event Channels. In *RTAS*, 2005.

Proceedings of the 13th IEEE Real Time and
Embedded Technology and Applications Symposium(RTAS'07)
0-7695-2800-7/07 $20.00 © 2007 IEEE