Report Number: WUCSE-2007-4

2007

Configurable Component Middleware for Distributed Real-Time Systems with Aperiodic and Periodic Tasks

Yuanfang Zhang, Christopher Gill, and Chenyang Lu

Many distributed real-time applications must handle mixed periodic and aperiodic tasks with diverse requirements. However, existing middleware lacks flexible configuration mechanisms needed to manage end-to-end timing easily for a wide range of different applications with both periodic and aperiodic tasks. The primary contribution of this work is the design, implementation and performance evaluation of the first configurable component middleware services for admission control and load balancing of aperiodic and periodic tasks in distributed real-time systems. Empirical results demonstrate the need for and effectiveness of our configurable component middleware approach in supporting different applications with periodic and aperiodic tasks.
... Read complete abstract on page 2.

Configurable Component Middleware for Distributed Real-Time Systems with Aperiodic and Periodic Tasks

Yuanfang Zhang, Christopher Gill, and Chenyang Lu

Complete Abstract:

Many distributed real-time applications must handle mixed periodic and aperiodic tasks with diverse requirements. However, existing middleware lacks flexible configuration mechanisms needed to manage end-to-end timing easily for a wide range of different applications with both periodic and aperiodic tasks. The primary contribution of this work is the design, implementation and performance evaluation of the first configurable component middleware services for admission control and load balancing of aperiodic and periodic tasks in distributed real-time systems. Empirical results demonstrate the need for and effectiveness of our configurable component middleware approach in supporting different applications with periodic and aperiodic tasks.

2007-4

Configurable Component Middleware for Distributed Real-Time Systems with Aperiodic and Periodic Tasks

Authors: Yuanfang Zhang, christopher Gill and Chenyang Lu

Corresponding Author: {yfzhang, cdgill, lu}@cse.wustl.edu

Type of Report: Other

Configurable Component Middleware for Distributed Real-Time Systems with Aperiodic and Periodic Tasks

Yuanfang Zhang, Christopher Gill and Chenyang Lu
Washington University, St. Louis, MO, USA

Abstract

Many distributed real-time applications must handle mixed periodic and aperiodic tasks with diverse requirements. However, existing middleware lacks flexible configuration mechanisms needed to manage end-to-end timing easily for a wide range of different applications with both periodic and aperiodic tasks. The primary contribution of this work is the design, implementation and performance evaluation of the first configurable component middleware services for admission control and load balancing of aperiodic and periodic tasks in distributed real-time systems. Empirical results demonstrate the need for and effectiveness of our configurable component middleware approach in supporting different applications with periodic and aperiodic tasks.

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 may be generated when a series of periodic sensor readings meets certain hazard detection criteria. This alert must be processed on multiple processors within an end-to-end deadline, for example to ensure that an industrial process is put into a fail-safe mode. User inputs and sensor readings may trigger various other real-time aperiodic tasks.

While there exist significant theoretical results on aperiodic scheduling [18], there is an increasing need to apply those results to the standards-based middleware that is increasingly being used for developing distributed real-time applications. In previous work we developed the first middleware-layer online admission control service supporting both aperiodic and periodic end-to-end tasks [24]. However, an admission controller with a single fixed acceptance strategy can not serve all applications well. Moreover, it is difficult to configure our earlier middleware to support different admission control strategies. Changing the supported strategy requires manually changing the code of the middleware, which is a challenging task due to the complexity and rich features of standard-based middleware such as TAO [11]. The lack of flexibility makes one implementation suitable for a specific set of applications (i.e., our previous implementation was targeted at a particular type of shipboard computing environment), but not for others. This is an important limitation given the extreme diversity of distributed real-time applications.

For an admission control service supporting, e.g., aperiodic utilization bounds [1], there are many possible strategies, the effectiveness of which depends significantly on workload characteristics and application requirements. For end-to-end periodic tasks, one common strategy for admission control is to perform an admission test the first time a periodic task arrives. Once a periodic task passes the admission test, all its jobs are allowed to be released immediately when they arrive. This is the traditional reservation model which is proper for critical periodic safety tests, such as oil tank leak detection.

However, this strategy improves middleware efficiency at the cost of increasing the pessimism of the admission test when the aperiodic utilization bound analysis is used for schedulability analysis, as the system effectively reserves a portion of the utilization for each periodic task. If a periodic task supports job skipping, such as when displaying non-critical sensor data, so as to reduce that pessimism we may perform the admission test for each job of the periodic task. Making this choice configurable allows system developers to choose the strategy that best suits their particular application requirements. Another configurable strategy is whether load balancing is performed. If more than one processor can execute a task, we can assign tasks to different processors at run time to distribute the workload adaptively, which may effectively increase the number of accepted tasks, but it requires replicating application code (and possibly mechanisms to maintain consistent states during task reallocation) on multiple processors.

To enhance the flexibility of middleware support for diverse applications with aperiodic and periodic tasks, we

have developed a set of new admission control services and load balancing services as configurable middleware components. QoS-aware component middleware platforms, such as CIAO [10], can be used to integrate these components together to form a component assembly for a particular application with its own requirements. Furthermore, we provide a new configuration pre-parser to help application developers easily configure these services to obtain desired behavior.

Research Contributions: In this paper, we have (1) classified applications with both aperiodic and periodic end-to-end tasks according to their characteristics and related them to suitable admission control and load balancing strategies; (2) developed what is to our knowledge the first example of component middleware supporting multiple admission control and load balancing strategies for both aperiodic and periodic end-to-end tasks; (3) developed a novel component configuration pre-parser and interfaces to select and configure services and attributes flexibly at system deployment; and (4) conducted empirical evaluations on a Linux testbed which demonstrate the effectiveness of our approach in supporting configurable services for both aperiodic and periodic end-to-end tasks, efficiently in middleware. Our work thus significantly enhances the flexibility and generality of distributed real-time middleware for aperiodic and periodic tasks.

Section 2 provides background information on aperiodic scheduling approaches and introduces the services provided by the CIAO and DAnCE middleware frameworks which our work extends. Section 3 presents our component middleware architecture, configurable strategies, and configurable component implementations for supporting aperiodic task scheduling end-to-end in distributed real-time systems. Section 4 describes our new configuration engine, which can flexibly configure different strategies for our services according to each application's requirements. Section 5 evaluates the performance of our approach. Finally, we offer concluding remarks in Section 6.

2 Background

Aperiodic Scheduling: 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 [22, 19, 12, 20, 16, 17, 13, 4, 21], and new schedulability tests based on aperiodic utilization bounds [1] and a new admission control approach [3] were introduced recently. In [24], we implemented and evaluated services for two suitable aperiodic scheduling techniques (aperiodic utilization bound [1] and deferrable server [22]). Since aperiodic utilization bound (AUB) has a comparable performance to deferrable server, and requires less complex scheduling mechanisms in middleware, we focus on the AUB scheduling technique in this paper. Our experience with AUB shows how configurability of other scheduling techniques can be integrated within real-time component middleware in a similar way.

We assume each aperiodic task has only one job, while a periodic task releases jobs periodically. According to AUB analysis [1], 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 AUB 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 AUB, the set of *current* tasks $S(t)$ at any time t is defined as the set of tasks that have been 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* at time t, $U_j(t)$, is defined as the sum of individual subtask utilizations on the processor, accrued over all current tasks. Under EDMS task T_i will meet its deadline if the following condition holds [1]:

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

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 [1]. When a processor becomes idle, the contribution of all completed aperiodic subtasks to the processor's synthetic utilization is removed.

Component Middleware: Component middleware platforms are an effective way of achieving customizable reuse of software artifacts. In these platforms, *components* are units of implementation and composition that collaborate with other components via *ports*. Groups of related components are connected together via their ports to form component assemblies. The ports define components' collaborations in terms of provided and required interfaces, event sources and sinks, or attributes. The ports isolate the components' contexts from their actual implementations. Component middleware platforms configure and deploy component assemblies, and provide execution environments and common middleware services.

Conventional component middleware platforms do not provide real-time quality of service (QoS) support. There have been several efforts to introduce of QoS in distributed systems. The FIRST Scheduling Framework (FSF) [2] proposes to compose several applications and to schedule the

available resources flexibly while guaranteeing hard real-time requirements. A real-time component type model [5], which integrates QoS facilities into component containers also was introduced based on the EJB and RMI specifications. A schedulability analysis algorithm [14] for hierarchical scheduling systems has been introduced for dependent components which interact through remote procedure calls. None of these approaches provides the configurable services for mixed periodic and aperiodic end-to-end tasks offered by our approach.

The Component-Integrated ACE ORB (CIAO) [10] is a CORBA Component Model (CCM) implementation built atop the TAO [11] real-time CORBA object request broker (ORB). CIAO supports real-time QoS by combining the flexibility of component middleware with the predictability of Real-time CORBA. CIAO abstracts DRE-critical systemic aspects such as real-time policies, as installable/configurable units. However, CIAO does not support aperiodic task scheduling, admission control and load balancing. We base our approach on CIAO to make configuring and managing aperiodic task support easier.

DAnCE [6] is a QoS-enabled component deployment and configuration engine which is compliant with the OMG Deployment and Configuration specification [15]. DAnCE parses component configuration and deployment descriptions and automatically configures ORBs, containers, and component server resources at system initialization time, to enforce end-to-end QoS requirements. DAnCE also simplifies the configuration, deployment, and management of common services used by applications and middleware, such as naming and event services. However, DAnCE does not provide tools to easily configure our admission control and load balancing services.

3 Component Middleware Approach

3.1 Architecture Overview

To support end-to-end aperiodic and periodic tasks, we have developed a new middleware architecture which extends CIAO to provide task management, and a front-end configuration engine for DAnCE. The key feature of our approach is a *configurable service framework* that can be customized for different sets of critical/non-critical and aperiodic/periodic tasks. Our framework is composed of *admission controller* (AC), *idle resetting* (IR) and *load balancer* (LB) components which interact with application components. The admission controller component provides online admission control and schedulability tests for tasks that arrive dynamically at run time. The load balancer component provides a task assignment plan to the admission controller, to which the admission controller applies its schedulability analysis. The IR component reports all completed subtasks on one processor to the AC component, so the AC component can remove their expected utilization to reduce the pessimism of the AUB analysis.

Figure 1 illustrates our distributed configurable component middleware architecture. All processors are connected by TAO's federated event channel [9]. We deploy one AC component and one LB component on a central task manager processor, and one IR component and one task effector (TE) component on each of multiple application processors. As an example, Figure 1 shows an end-to-end task T_i composed of 3 consecutive subtasks, $T_{i,1}$, $T_{i,2}$ and $T_{i,3}$, executing on other processors. $T_{i,1}$ and $T_{i,2}$ have duplicates on different application processors. When task T_i arrives at an application processor, the task effector component on that processor pushes a "Task Arrived" event to the AC component and holds the task until it receives an "Accept" command from the AC component. The AC component and LB component decide whether to accept the task, and if so, where to assign its subtasks. The solid line and the dashed line show two possible assignments of subtasks. If the first subtask $T_{i,1}$ is not assigned to the processor where T_i arrived, we call this assignment Task Reallocation.

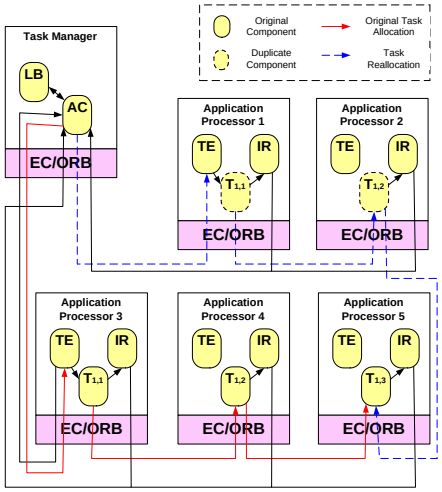

Figure 1. Distributed Middleware Architecture

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 AC components on multiple processors 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 the affected processors. A potential disadvantage of the centralized architecture is that the AC component may become a communication bottleneck and thus affect scalability. However, this is not a concern in systems in which processors are connected by high-speed real-time networks. Furthermore, the computation time of the schedulability analysis is

significantly lower than task execution times in many high performance real-time applications. In summary, while our Admission Control component approach can be deployed in a centralized or distributed fashion, we have focused first on a centralized approach that has lower complexity and overhead. As future work we plan to examine the benefits and costs of decentralized admission control.

3.2 Applications Categories and Middleware Strategies

Three criteria distinguish how different applications with aperiodic tasks should be supported. **C1:** whether application components are replicated on multiple processors. **C2:** whether persistent state must be preserved between jobs. **C3:** whether the application can tolerate job skipping.

According to these different application categories, the AC, IR and LB components can be configured to use different strategies. For each component, which strategy is more suitable depends on these criteria and the application's overhead requirements. As we discuss in Section 5, experiments we have run under different combinations of strategies can provide valuable configuration guidance to application developers. Moreover, we have designed all strategies as configurable component attributes, and provide a configuration pre-parser and a component configuration interface, to allow developers to select and configure the service's mechanisms and attributes flexibly, according to each application's specific needs. We now examine the different strategies for each component and the trade-offs among them.

3.2.1 Admission Control Strategies

Admission control offers significant advantages for scheduling systems with aperiodic and periodic tasks, by providing online schedulability guarantees to tasks arriving dynamically. Our AC component supports two different strategies: AC per Task and AC per Job. AC per Task performs that test whenever a task arrives while AC per Job performs the admission test whenever a job of a task arrives.
AC per Task: Considering the admission overhead and the fixed interarrival times of periodic tasks, one strategy is to perform an admission test only when a periodic task first arrives. Once a periodic task passes the admission test, all its jobs are allowed to be released immediately when they arrive. This strategy improves middleware efficiency at the cost of increasing the pessimism of the admission test. In the AUB analysis [1], the contribution of a job to the synthetic utilization of a processor can be removed when the job's deadline expires (or when the CPU idles if the resetting rule is used and the job has been completed). If admission control is performed only at task arrival time, however, the AC component must reserve the synthetic utilization of the task throughout its lifetime. As a result, it cannot reduce the synthetic utilization between the deadline of a job and the arrival of the subsequent job of the same task, which may cause pessimistic admission decisions [1].
AC per Job: If it is possible to skip a job of a periodic task, the alternative strategy to reduce pessimism is to apply the admission test to every job of a *periodic* task. This strategy is practical for many systems, since the AUB test is highly efficient when used for AC, as shown in Section 5.3 for our experiment results.

Thus, only applications satisfying criterion C3 are suitable for the second strategy. Moreover, the second strategy reduces pessimism at the cost of increasing overhead. The application developer thus needs to consider the tradeoff between overhead and pessimism when they decide the proper configuration.

3.2.2 Idle Resetting Strategies

As presented in [1, 24], the use of a resetting rule can reduce the pessimism of the AUB schedulability test significantly. There are three ways to configure IR components in our approach.
No IR: The first strategy is to use no resetting at all, so that if the subtasks complete their executions, the contribution of completed jobs to the processor's synthetic utilization is not removed until the task deadline. This strategy avoids the resetting overhead, but increases the pessimism of schedulability analysis.
IR per Task: The second strategy is that each IR component records completed aperiodic subtasks on one processor. Whenever the processor is idle, a lowest priority thread called an *idle detector* begins to run. Its main job is to report the completed *aperiodic* jobs to the AC component 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.
IR per Job: The third strategy is that each IR component records and reports not only the completed aperiodic subtasks but also the completed jobs of *periodic* subtasks.

The first of these three strategies avoids the resetting overhead, but is the most pessimistic. The third strategy removes the contribution of completed aperiodic and periodic tasks more frequently than other two strategies. Although it has the least pessimism, it introduces the most overhead. The second strategy is a tradeoff between the first and the third strategies.

3.2.3 Load Balancing Strategies

Under AUB-based AC, Load balancing can effectively improve system performance in the face of dynamic task arrivals [1]. We use a heuristic algorithm to assign subtasks to processors at run-time, which always assigns a subtask

to the processor with the lowest synthetic utilization among all processors with its replicas. [1] Since migrating a subtask between processors introduces extra overhead, when we accept a new task, we only decide the load balancing issue for this new task and do not change the assignment plan for any other task in the current task set. This service also has three strategies.

No LB: The first strategy does not perform load balancing. Each subtask is assigned to particular processor.

LB per Task: Each task will only be assigned once at its first arrival time. This strategy is suitable for applications which must maintain persistent state between any two consecutive jobs of a periodic task.

LB per Job: The third strategy is the most flexible one. All jobs from a periodic task are allowed to be assigned to different processors when they arrive.

The first strategy is more suitable for applications which can not satisfy criterion C1. The second strategy is most applicable for applications which only satisfy C1, but can not satisfy criterion C2. The third strategy is most suitable for applications which satisfy both C1 and C2.

3.2.4 Combining AC, IR and LB Strategies

When we use the AC, IR and LB components together, their strategies can be configured in 18 different combinations. However some combinations of the strategies are invalid. The AC-per-Task/IR-per-Job combination is not reasonable, because per job idle resetting means the synthetic utilizations of all completed jobs of periodic subtasks are to be removed from the central admission controller, but per task admission control requires that the admission controller reserves the synthetic utilization for all accepted periodic tasks, so an accepted periodic task does not need to go through admission control again before releasing its jobs. These two requirements are thus contradictory, and we can exclude the corresponding configurations as being invalid. Removing this invalid AC/IR combination means removing 3 invalid AC/IR/LB combinations, so there are only 15 reasonable combinations of strategies left. The advantage of our middleware architecture and configuration engine is that they allow application developers to configure middleware services to achieve any combination of strategies.

3.3 Component Implementation

Configurable component middleware standards, such as the CORBA Component Model (CCM), can help to reduce the complexity of developing large distributed applications by defining a component-based programming paradigm. They also help by defining a standard configuration framework for packaging and deploying reusable software components. The Component Integrated ACE ORB (CIAO) [23] is an implementation of CCM that is specifically designed and optimized for distributed real-time embedded (DRE) systems. To support the different strategies described in Section 3, and to allow flexible configuration of suitable combinations of those strategies for a variety of applications, we have integrated admission control, idle resetting and load balancing into CIAO as configurable components. Each component provides a specific service with configurable attributes and clearly defined interfaces for collaboration with other components, and can be instantiated multiple times with the same or different attributes. Component instances can be connected together at run time through appropriate ports to form a distributed real-time application. As shown in Figure 2, we have designed and implemented 6 configurable components to support distributed real-time aperiodic and periodic tasks end-to-end in CIAO. The implementation using C++ language is based on ACE/TAO/CIAO version 5.5.1/1.5.1/0.5.1.

Task Effector: holds the arriving tasks, waits for the AC component decision and releases tasks.

First/Intermediate Subtask: executes the first or an intermediate subtask at a given priority.

Last Subtask: executes the last subtask at a given priority.

Idle Resetting: records and reports the completed subtasks when a processor goes idle.

Admission Control: decides whether to accept tasks.

Load Balancing: computes task allocations so as to balance the processors' synthetic utilizations.

Each component may have several configurable attributes, so that it can be instantiated with different properties, like its criticality and execution time (for application components) or its strategy (for AC, IR and LB components). Our admission control and load balancing approaches adopt a centralized architecture, which employs one instance of an Admission Control component and one instance of a Load Balancing component running on a central processor (called the "Task Manager").

Each application processor contains one instance of a Task Effector component and one instance of an Idle Resetting component. The Task Effector component on each processor reports the arrival of tasks on that processor to the Admission Control component which then releases or rejects the tasks based on the admission control decision. The Idle Resetting component on each application processor records and reports the completed subtasks on that processor to the Admission Control component, whenever that processor goes idle. Each end-to-end task is implemented by a chain of First/Intermediate Subtask components and one Last Subtask component. Figures 1 and 2 show the structure of, and relationships among, these components.

[1] The focus here is not on load balancing algorithms. Our configurable middleware may be easily extended to incorporate other load balancing algorithms according to each application's needs.

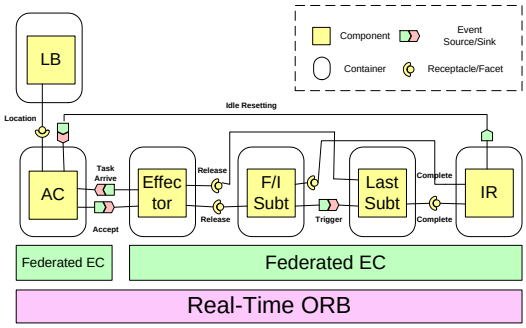

Figure 2. Component Implementation

Task Effector Component: When a task arrives, the Task Effector component puts it into a waiting queue and pushes a "Task Arrival" event to the AC component. When it receives an "Accept" event from the AC component, a task waiting in the queue will be released immediately. The Task Effector component has two configurable attributes. One is a processor ID, which is used to distinguish Task Effector component instances deployed on different processors. The other is the AC-per-job/AC-per-task attribute, which indicates whether periodic tasks are admitted per job or per task. If the periodic tasks are admitted per job, then before releasing any job of a periodic task the Task Effector component will hold it until receiving an "Accept" event from the AC component. Otherwise, once a job of a periodic task is admitted, the AC component reserves CPU capacity for it, so all subsequent jobs from that same periodic task can be released immediately without going through the AC component again. These attributes can be set at the creation of the component instances and may be modified at run time.

First/Intermediate (F/I) Subtask Component and Last Subtask Component: Both the F/I and Last Subtask components execute application subtasks. The only difference between these two kinds of components is that the First/Intermediate Subtask Component has an extra port that publishes "Trigger" events to initiate the execution of the next subtask. The Last Subtask Component does not need this port, since the last subtask does not have a next subtask. Both of these kinds of components contain a dispatching thread which executes a particular subtask at a specified priority. Both kinds of components have three configurable attributes. The first two attributes are execution time and priority level, which are normally set at the creation of the component instances as specified by application developers. The third attribute is No-IR, IR-per-task, or IR-per-job, which means the resetting rule either is not enabled or is enabled per task or per job respectively. Per-task means the Idle Resetting Component will not be notified when periodic subtasks complete. Since each job of an aperiodic task can be treated as an independent aperiodic task with one release, the idle resetting component is notified when aperiodic subtasks complete. The dispatching threads in a First/Intermediate Subtask Component or a Last Subtask Component are triggered by either a "Release" method call from the Task Effector Component or a "Trigger" event from a previous First/Intermediate Subtask Component. Both First/Intermediate Subtask and Last Subtask components call the "Complete" method of the local Idle Resetting component when a subtask completes.

Idle Resetting (IR) Component: It receives "Complete" method calls from local First/Intermediate or Last Subtask components, and pushes "Idle Resetting" events to the Admission Control component. It has one attribute, the processor ID, which is used to distinguish the component instances sitting on different processors.

Admission Control (AC) Component: It consumes "Task Arrival" events from the Task Effector components and "Idle Reset" events from the Idle Resetting components. It publishes "Accept" events to the Task Effector components to allow task release. It makes "Location" method calls on the Load Balancing component to ask for proposed task assignment plans. The Admission Control component has an No-LB/LB-per-task/LB-per-job attribute, which indicates whether load balancing is turned on or off, and if it is enabled whether it is per task or per job. If that attribute is set to T, once a periodic task is admitted, its subtask assignment is decided and kept for all following jobs. However, aperiodic tasks do not have this restriction as they are only allocated at their single job arrival time. A value of J means the subtask assignment plan can be changed for each job of an accepted task.

Load Balancing (LB) Component: It receives "Location" method calls from the AC component, which ask for assignment plans for particular tasks. The Load Balancing component tries to balance the synthetic utilization among all processors, and may modify a previous allocation plan when a new task is accepted. It returns an assignment plan that is acceptable and attempts to minimize the difference of synthetic utilizations on all processors after accepting that task. Alternatively, the LB component may tell the AC component that the system would be unschedulable if the task were accepted.

4 Deployment and Configuration

Although we have designed our configurable components specifically for developers who want middleware support for aperiodic scheduling, it is still not easy for an application developer to assemble and deploy those components correctly by hand. Therefore, we have automated the deployment and configuration of these components using standards-based component middleware techniques in CIAO. CIAO's realization of the OMG's Deployment and Configuration specification [15] is called the Deployment

and Configuration Engine (DAnCE) [6]. DAnCE can translate an XML-based assembly specification into the execution of deployment and configuration actions needed by an application. Assembly specifications are encoded as descriptors which describe how to build distributed applications using available component implementations. Information contained in the descriptors includes the number of processors, what component implementations to use, how and where to instantiate components, and how to connect component instances in an application.

It is error prone for an application developer to write those descriptors by hand. Although tools such as CoSMIC [8] are provided to help generate those XML files, those tools do not consider the configuration requirements of the new services we have created. We therefore provide a specific configuration engine that acts as a front-end to DAnCE, to configure our aperiodic scheduling services for application developers who require configurable aperiodic scheduling support in component middleware.

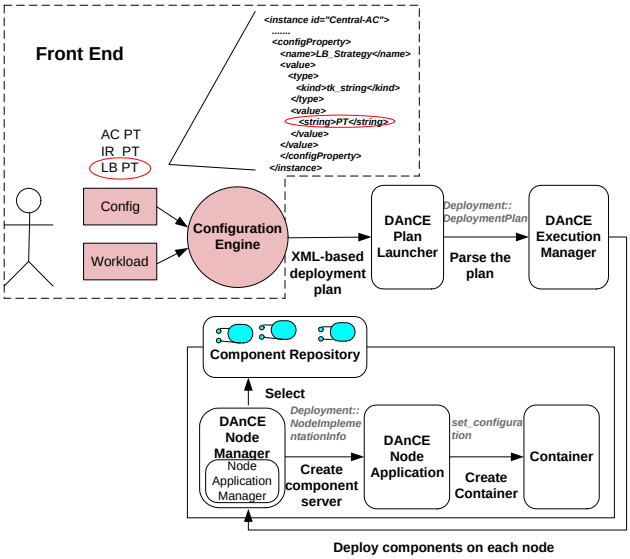

Figure 3. Dynamic Configuration Process

Front-end Configuration Engine: As is shown in Figure 3, the application developer first provides two text files. One is a configuration file, which consists of the configuration requirements for the admission control, idle resetting and load balancing services. The other is a workload file which describes each end-to-end task and where its subtasks execute. Our front-end configuration engine parses these two files, then generates an XML-based deployment plan, which can be recognized by DAnCE. As an example, Figure 3 shows a configuration file which sets the AC, IR and LB services to per-task (PT). Figure 3 also shows a part of the XML file generated by our configuration engine, which shows the LB strategy (LB_Strategy) as PT.

To enforce end-to-end deadline monotonic scheduling, the First/Intermediate Subtask and Last Subtask components both expose an attribute called "priority". When our configuration engine reads the workload file, it assigns priorities in order of tasks' end-to-end deadlines, and writes this priority information into the generated XML deployment plan, to be parsed by DAnCE later. Our front-end configuration engine not only generates well formed assembly specifications, according to the application developers' instructions, but it also performs a feasibility check on the configuration file, to ensure correct handling of dependent constraints. For example, per task admission control with per job idle resetting would be contradictory as we mentioned in Section 3. Since a developer might specify incompatible service configuration combinations, our approach can detect and disallow them. Finally, if no configuration file is provided or it omits configuration information, the system will use default configuration settings, i.e., per task admission control, idle resetting and load balancing.

DAnCE: We have used the <configproperty> feature of DAnCE to extend the set of attributes configured flexibly according to other configuration settings. For example, if the load balancing service is configured using the per-task strategy, the corresponding property of the AC component should also be set to per-task.

DAnCE's Plan Launcher parses the XML-based deployment plan and stores the property name (LB_Strategy) and value in a data structure (Property) which is a field of the AC instance definition structure. The definitions of the AC instance and all other component instances comprise a deployment plan (Deployment::DeploymentPlan) that is then passed to DAnCE's Execution Manager for execution. The Execution Manager propagates the deployment plan data structure to DAnCE's Node Application Manager, which parses it into an initialization data structure (NodeImplementationInfo). Finally, the Node Application Manager passes the initialization data structure to the Node Application. When the Node Application installs the AC component instance, it also initializes the LB_Strategy attribute of the AC component through a standard Configurator interface (set_configuration), using the initialization data structure it received.

5 Experimental Results

To validate our approach, and to evaluate the performance, overheads and benefits resulting from it, we conducted a series of experiments which we describe in this section. The experiments were performed on a testbed consisting of six machines connected by a 100Mbps Ethernet switch. Two are Pentium-IV 2.5GHz machines with 1G RAM and 512K cache each, two are Pentium-IV 2.8GHz machines with 1G RAM and 512K cache each, and the other

two are Pentium-IV 3.40GHz machines with 2G RAM and 2048K cache each. Each machine runs version 2.4.22 of the Redhat Linux operating system. One of the Pentium-IV 2.5GHz machines is used as a central task manager where the admission control and load balancing components are deployed. The other five machines are the application processors on which task effector, subtask and idle resetting components are deployed.

5.1 Random Workloads

We first randomly generated 10 sets of 9 tasks, each including 4 aperiodic tasks and 5 periodic tasks. The number of subtasks per task is uniformly distributed between 1 and 5. Subtasks are randomly assigned to 5 application 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. Each subtask is assigned to a processor, and has a duplicate sitting on a different processor which is randomly picked from the other 4 application processors.

In Section 3, we showed 15 reasonable combinations of strategies. In this experiment, we only evaluated the five most representative combinations of strategies, ran 10 task sets using each combination and compared them. The other 10 reasonable combinations can be evaluated in a similar way using our configurable components. In the following figures, a three element tuple denotes a combination of settings for the three configurable services. The first element refers to the admission control service. The second element refers to the idle resetting service. The third element refers to the load balancing service.

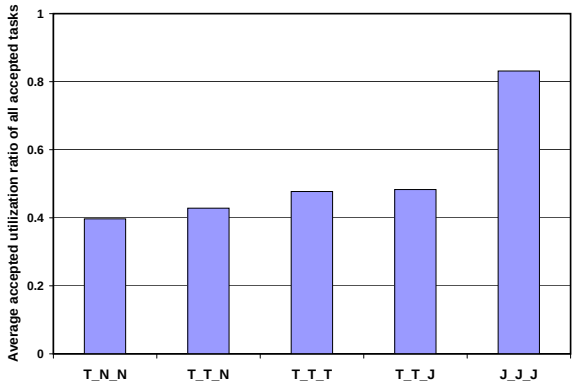

Figure 4. Accepted Utilization Ratio

The performance metric we used in these evaluations is the *accepted utilization ratio*, i.e., the total utilization of jobs accepted by the admission controller divided the total utilization of all jobs requesting admission. To be concise, we use capital letters to represent strategies.

N: a service is **not** enabled in this configuration.
T: a service is enabled for each **task**.
J: a service is enabled for each **job** of a task.

The bars in Figure 4 show the average results over the 10 task sets. As is shown in Figure 4, enabling either the idle resetting service or the load balancing service can increase the utilization of tasks admitted. Moreover, the experiment shows that enabling all three services per job (J_J_J) significantly outperforms the configuration which enables the three services per task (T_T_T), even though the J_J_J configuration introduces more overhead. We also notice the difference is small when we only change the configuration of the load balancing component from per task to per job and keep the configuration of other two services the same. This is because when we randomly generate these 10 task sets, the synthetic utilization of each processor is similar. That feature of our first experiment greatly reduced the improvement due to the load balancing component since the original task set is well balanced. To show the potential benefit of the Load Balancing component, we designed another experiment which we describe in the next section.

5.2 Imbalanced Workloads

In the second experiment, we use an imbalanced workload. It is representative of dynamic systems in which a subset of the system may experience heavy load. For example, the arrival of a large number of targets may cause sharp increase of the load on the target recognition subsystem. In this experiment, we divided the 5 application processors into two groups. One group contains 3 processors hosting all tasks. The other group contains 2 processors hosting all duplicates. 10 task sets are randomly generated as in the above experiment, except that all subtasks were randomly assigned to 3 application processors in the first group and the number of subtasks per task is uniformly distributed between 1 and 3. The synthetic utilization for any of these three processors is 0.7. Each subtask has one duplicate sitting on one processor in the second group.

For the experimental runs represented by the three bars in the middle of Figure 5, we kept the admission control and the idle resetting strategies the same (per task), but changed the load balancing strategy from none to per task, then to per job. 10 task sets were run 3 times for the 3 different combinations, and for each combination we then averaged the utilization acceptance ratio over the 10 results. As figure 5 shows, load balancing per task provides a significant improvement when compared with the results without load balancing. However there is not much difference between load balancing per task and load balancing per job.

From the above two experiments, we found that application characteristics can really impact the performance of different strategies. Our design of the AC, IR and LB ser-

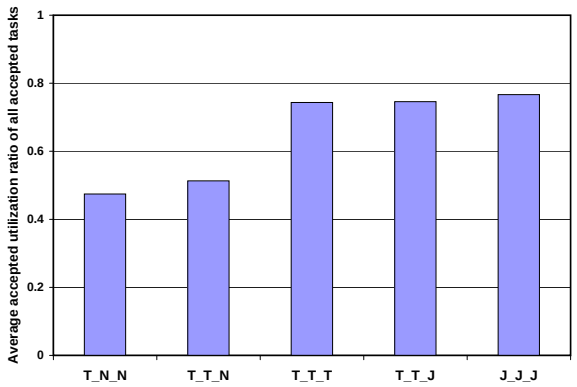

Figure 5. Load Balancer Strategy Comparison

vices as easily configurable components allows application developers to select configurations based on the characteristics and requirements of their applications, and these results.

5.3 Overhead of Services

To evaluate the efficiency of our component-based middleware services, we used 3 processors to run applications and another processor to run the admission control service and load balancing components. The workload is randomly generated in the same way as described in Section 4, except that the number of subtasks per task is uniformly distributed between 1 and 3. Each experiment ran for 5 minutes. We examined the different sources of overhead that may occur when a task arrives at Task Effector component TE1, after which AC and LB components run it from component TE1 or re-allocate it to Task Effector component TE2. Figure 6 shows how the total delay for each service includes the costs of operations located in several components. Table 1 lists the operation numbers shown in Figure 6 to provide a detailed accounting of the delays resulting from different combinations of services.

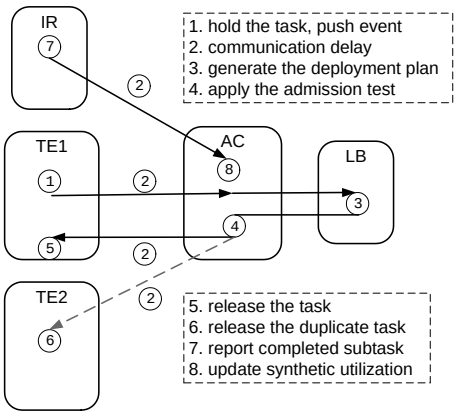

Figure 6. Sources of Overhead/Delay

	mean	max
AC without LB (1+2+4+2+5)	1114	1248
AC with LB (1+2+3+4+2+5) (no re-allocation)	1116	1253
AC with LB (1+2+3+4+2+6) (re-allocation)	1201	1327
LB (no re-allocation) (1+2+3+2+5)	1113	1250
LB (re-allocation) (1+2+3+2+6)	1198	1309
IR (on AC side) (8)	17	18
IR (other part) (7+2)	662	683
Communication Delay (2)	322	361

Table 1. Overhead of Services (μs)

To calculate the delays for AC without LB, AC with LB without re-allocation and LB without re-allocation, we can simply calculate the interval between when one task arrives on a processor and when the task is released on the processor. However, if the load balancing service re-allocates the first subtask on a different processor using its duplicate, as in the case of AC with LB, it is difficult to determine a precise time interval between when one task arrives on one processor and when it is released on another processor, because like many of the systems for which our approach is suitable, our experiment environment does not provide sufficiently high resolution time synchronization among processors. We therefore measure the overheads on all involved processors individually, then add them together plus twice the communication delays (step 2 in Figure 6) between the processors. Three processors are involved: the processor where the task arrives (step 1), the central admission control processor (steps 3 and 4) and the other processor where the duplicate task is released (step 6). We ran this experiment on KURT-Linux [7] version 2.4.22, which provides a CPU-supported timestamp counter with nanosecond resolution. By using instrumentation provided with the KURT-Linux distribution, we can obtain a precise accounting of operation start and stop times and communication delays. To measure the communication delay between the application processor and the admission control processor on our experimental platform, we pushed an event back and forth between the application processor to the admission control processor 1000 times, then calculated the mean and max value among 1000 results. We then divided the round trip time by 2 to obtain the approximate mean and maximum communication delays between the application processor and the admission control processor.

The total delay for the load balancing service, when reallocation happens, is measured in the same way as for the case of AC with LB with reallocation. To calculate the delay from the idle resetting service, we divide its execution into two parts. The small overhead on the admission control processor must be counted in the overall delay. How-

ever, the large overhead on the application processor and the communication delay only happen during CPU idle time, and although it represents an additional overhead induced by the IR service, it does not affect performance, which is why we report the two parts separately in Table 1. From the results in Table 1, we can see that all of the delays induced by our configurable components are less than 2 ms, which is acceptable to many distributed real-time system environments. However for applications with tight schedules, a developer can make decisions on how to configure the services based on this delay information as well as on the effects of the different configurations on task management, which we discussed in Section 3.2.3.

6 Conclusions

The work presented in this paper represents a promising step toward developing configurable admission control and load balancing support for different kinds of distributed applications with aperiodic and periodic tasks in real-time component middleware. We have designed and implemented effective configurable middleware components that provide online admission control and load balancing and can be easily configured and deployed on different processors. Our configuration engine can automatically process the user's configuration file and generate a corresponding deployment plan for DAnCE, thus making it easier for developers to select suitable configurations, and to avoid invalid ones. Empirical results we obtained showed that (1) our configurable component middleware are well suited for satisfying different applications with a variety of alternative characteristics and requirements and (2) our component middleware services are highly efficient on a Linux platform.

References

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

[2] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. Palencia, and M. Trimarchi. FSF: A Real-Time Scheduling Architecture Framework. In *RTAS*, 2006.

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

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

[5] M. A. de Miguel. Integration of QoS Facilities into Component Container Architectures. In *ISORC*, 2002.

[6] G. Deng, D. C. Schmidt, C. Gill, and N. Wang, editors. *QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems*. CRC Press. to appear.

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

[8] A. Gokhale. Component Synthesis using Model Integrated Computing. `www.dre.vanderbilt.edu/cosmic`, 2003.

[9] 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.

[10] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

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

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

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

[14] J. L. Lorente, G. Lipari, and E. Bini. A Hierarchical Scheduling Model for Component-Based Real-Time Systems. In *WPDRTS*, 2006.

[15] Object Management Group. *Deployment and Confi guration Specifi cation*, OMG Document ptc/2003-07-02 edition, July 2003.

[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] L. Sha and et. al. Real Time Scheduling Theory: A Historical Perspective. *The Journal of Real-Time Systems*, 10:101–155, 2004.

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

[20] 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.

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

[22] 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.

[23] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring Real-time Aspects in Component Middleware. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[24] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems. In *RTAS*, 2007. to appear.