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

Yuanfang Zhang, Christopher D. Gill, *Member*, *IEEE*, and Chenyang Lu, *Member*, *IEEE*

Abstract—Different distributed real-time systems (DRS) must handle aperiodic and periodic events under diverse sets of requirements. While existing middleware such as Real-Time CORBA has shown promise as a platform for distributed systems with time constraints, it lacks flexible configuration mechanisms needed to manage end-to-end timing easily for a wide range of different DRS with both aperiodic and periodic events. 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 event handling in DRS. Empirical results demonstrate the need for, and the effectiveness of, our configurable component middleware approach in supporting different applications with aperiodic and periodic events, and providing a flexible software platform for DRS with end-to-end timing constraints.

Index Terms—Component middleware, dynamic real-time task allocation, load balancing and admission control.

1 Introduction

Many distributed real-time systems (DRS) must handle a mix of aperiodic and periodic events, including aperiodic events with end-to-end deadlines whose assurance is critical to the correct behavior of the system. Requirements for increased software productivity and quality motivate the use of open distributed object computing (DOC) middleware such as CORBA, rather than building applications entirely from scratch using proprietary methods. The use of CORBA middleware has increased significantly in DRS domains such as aerospace, telecommunications, medical systems, distributed interactive simulations, and computer-integrated manufacturing, which are also characterized by stringent quality of service (QoS) requirements [1]. 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, e.g., to put an industrial process into a fail-safe mode. User inputs and other sensor readings may trigger other real-time aperiodic events.

While traditional real-time middleware solutions such as Real-Time CORBA [2] and Real-Time Java [3] have shown promise as distributed software platforms for systems with time constraints, existing middleware systems lack the flexibility needed to support DRS with diverse application semantics and requirements. For example, load balancing is an effective mechanism for handling variable real-time workloads in a DRS. However, its suitability for DRS highly depends on their application semantics. Some digital control algorithms (e.g., proportional-integral-derivative control) for physical systems are stateful and hence not amenable for frequent task reallocation caused by load balancing, while others (e.g., proportional control) do not have such limitations. Similarly, job skipping (skipping the processing of certain instances of a periodic task) is useful for dealing with transient system overload. While job skipping is not suitable for certain critical control applications in which missing one job may cause catastrophic consequences on the controlled system, other applications ranging from video reception to telecommunications may be able to tolerate varying degrees of job skipping [4].

Therefore, a key open challenge for DRS is to develop a flexible middleware infrastructure that can be easily configured to support the diverse requirements of different DRS. Specifically, middleware services such as load balancing and admission control must support a variety of alternative *strategies* (algorithms and inputs corresponding to those algorithms). Furthermore, the configuration of those strategies must be supported in a flexible yet principled way, so that system developers are able to explore alternative configurations without choosing invalid configurations by mistake.

Providing middleware services with configurable strategies, thus, faces several important challenges: 1) services must be able to provide configurable strategies, and configuration tools must be added or extended to allow configuration of those strategies; 2) the specific criteria that distinguish which service strategies are preferable must be identified, and applications must be categorized according to those criteria; and 3) appropriate combinations of services' strategies must be identified for each such application category, according to its characteristic criteria. To address these challenges, and thus to enhance support

- *Y. Zhang is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: yfzhang@cse.wustl.edu.*
- *C.D. Gill and C. Lu are with the Department of Computer Science and Engineering, Washington University, Campus Box 1045, One Brookings Drive, St. Louis, MO 63130-4899. E-mail: {cdgill, lu}@cse.wustl.edu.*

Manuscript received 9 May 2007; revised 11 Feb. 2009; accepted 31 Mar. 2009; published online 17 Apr. 2009.
Recommended for acceptance by M. Yamashita.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-05-0174.
Digital Object Identifier no. 10.1109/TPDS.2009.67.

for diverse DRS with aperiodic and periodic events, we have designed and implemented a new set of component middleware services including end-to-end event scheduling, admission control, and load balancing. We have also developed configuration tools to integrate these service components for each particular application according to its specific criteria.

Research contributions. In this work, we have

1. developed what is to our knowledge the first set of configurable component middleware services supporting multiple admission control and load balancing strategies for handling aperiodic and periodic events;
2. developed a novel component configuration preparser and interfaces to configure real-time admission control and load balancing services flexibly at system deployment time;
3. defined categories of distributed real-time applications according to specific characteristics, and related them to suitable combinations of strategies for our services; and
4. provided a case study that applies different configurable services to a domain with both aperiodic and periodic events, offers empirical evidence of the overheads involved and the trade-offs among service configurations, and demonstrates the effectiveness of our approach in that domain.

Our work, thus, significantly enhances the applicability of real-time middleware as a flexible infrastructure for DRS.

Section 2 introduces the middleware systems and scheduling theory underlying our approach. Sections 3-5 present our middleware architecture, configurable strategies, and component implementations for supporting end-to-end event handling in DRS. Section 6 describes our new configuration engine extensions, which can flexibly configure different strategies for our services according to each application's requirements. Section 7 evaluates the performance of our approach, including trade-offs among different service strategy combinations, and characterizes the overheads introduced by our approach. Section 8 presents a survey of related work, and we offer concluding remarks in Section 9.

2 Background

Task model. We consider DRS comprised of physical systems generating aperiodic and periodic events that must be processed on distributed computing platforms subject to end-to-end deadlines. Henceforth, the processing of a sequence of related events is referred to as a *task*. A task T_i is composed of a chain of *subtasks* $T_{i,j}(1 \leq j \leq n_i)$ located on different processors. The first subtask $T_{i,1}$ of a task T_i is triggered by a periodic timer event or an aperiodic event generated by the system. Upon completion, a subtask $T_{i,j}$ pushes another event which triggers its successor subtask $T_{i,j+1}$. Each subtask of a periodic task is a sequence of subjobs. Each periodic task is a sequence of *jobs* with each job being a chain of *subjobs* of each of the task's subtasks. The arrival time of a job or subjob is when it becomes available for execution. The release time of a job or subjob occurs after its arrival, following its admission by the admission controller when it is released for execution by the system. Every job of a task must be completed within an end-to-end deadline that is its maximum allowable response time. The period of a periodic task is the interarrival time of consecutive subjobs of the first subtask of the periodic task. An aperiodic task does not have a period. The interarrival time between consecutive subjobs of its first subtask may vary widely and, in particular, can be arbitrary small. The worst-case execution time of every subtask, the end-to-end deadline of every task, and the period of every periodic task in the system are known.

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*. The ports isolate the components' contexts from their actual implementations. Component middleware platforms provide execution environments and common services, and support additional tools to configure and deploy the components.

In previous work, we developed the first instantiation of a middleware admission control service supporting both aperiodic and periodic events [5] (on TAO, a widely used Real-Time CORBA middleware). However, our previous admission control service only included a fixed set of strategies. As is shown in Section 4, a more diverse and configurable set of interoperating services and service strategies is needed to support DRS with different application semantics. Unfortunately, it is difficult to extend implementations that rely directly on distributed object middleware, such as our original admission control service. Specifically, in those middleware systems changing the supported strategy requires explicit changes to the service code itself, which can be tedious and error prone in practice.

The Component-Integrated ACE ORB (CIAO) [6] implements the Light Weight CORBA Component Model (CCM) specification [7] and is built atop the TAO [8] real-time CORBA object request broker (ORB). CIAO abstracts common real-time policies as installable and configurable units. However, CIAO does not support aperiodic task scheduling, admission control, or load balancing. To develop a flexible infrastructure for DRS, in this work, we develop new admission control and load balancing services, each with a set of alternative service strategies on top of CIAO. Furthermore, we extended CIAO to configure and manage both services.

DAnCE [9] is a QoS-enabled component deployment and configuration engine that implements the Object Management Group (OMG)'s Light Weight CCM Deployment and Configuration specification [7]. DAnCE parses component configuration/deployment descriptions and automatically configures and deploys ORBs, containers, and server resources at system initialization time, to enforce end-to-end QoS requirements. However, DAnCE does not provide certain essential features needed to configure our admission control and load balancing services correctly, e.g., to disallow invalid combinations of our service strategies.

Aperiodic scheduling. Aperiodic tasks have been studied extensively in real-time scheduling theory, including work on aperiodic servers that integrate scheduling of aperiodic

and periodic tasks [10]. New schedulability tests based on aperiodic utilization bounds (AUBs) [11] and a new admission control approach [12] also were introduced recently. In our previous work [5], we implemented and evaluated admission control services for two suitable aperiodic scheduling techniques (aperiodic utilization bound [11] and deferrable server [13]) on TAO. Since AUB has comparable performance to deferrable server, and requires less complex scheduling mechanisms in middleware, we focus exclusively on the AUB technique in this paper. Our experiences with AUB reported in this paper show how configurability of other techniques can be integrated within real-time component middleware in a similar way.

With the AUB approach, three kinds of service strategies must be made configurable to provide flexible and principled support for diverse DRS with aperiodic and periodic tasks: 1) when admissibility is evaluated (to trade off the granularity and, thus, the pessimism of admission guarantees), 2) when the contributions of completed subjobs of subtasks can be removed from the schedulability analysis used for admission control (to improve accuracy of the schedulability analysis and, thus, reduce pessimistic denials of feasible tasks), and 3) when jobs of tasks can be assigned to different processors (to balance load and improve system performance).

In AUB [11], the set of *current* tasks $S(t)$ at any time t is defined as the set of tasks that have released jobs but whose deadlines have not expired. Hence, $S(t) = \{T_i \mid A_i \leq t < A_i + D_i\}$, where A_i is the release time of the first subjob of the current job for task T_i, and D_i is the relative deadline of the current job 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. According to AUB analysis, a system achieves its highest schedulable synthetic utilization bound under the End-to-end Deadline Monotonic Scheduling (EDMS) algorithm under certain assumptions. Under EDMS, a subtask has a higher priority if it belongs to a task with a shorter end-to-end deadline. Note that AUB does not distinguish aperiodic from periodic tasks. All tasks are scheduled using the same scheduling policy. Under EDMS task T_i will meet its deadline if the following schedulability condition holds [11]:

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

where V_{ij} is the jth processor that task T_i visits. A task (or an individual job) can be admitted only when this condition continues to be satisfied for all currently admitted tasks and this task. Since applications may or may not tolerate job skipping, whether this condition is checked only when the first job of a task arrives or whenever each job arrives should be configurable.

According to the definition of the current task set in AUB, a task remains in the current task set even if it has been completed, as long as its deadline has not expired. To reduce the pessimism of the AUB analysis, a *resetting rule* is introduced in [11]. When a processor becomes idle, the contribution of all completed subjobs to the processor's synthetic utilization can be removed without affecting the correctness of the schedulability condition (inequality 1). Since the resetting rule introduces extra overhead, whether the contribution of only completed aperiodic subjobs or of both completed aperiodic and periodic subjobs can be removed early should be made configurable. Under AUB-based schedulability analysis, load balancing also can effectively improve system performance [11]. However, some applications require persistent state preservation between jobs of the same task, so it also should be made configurable whether a task can be reallocated to a different processor for each job.

3 Middleware Architecture

To support end-to-end aperiodic and periodic tasks in diverse distributed real-time applications, we have developed a new middleware architecture. The key feature of our approach is a *configurable component framework* that can be customized for different sets of aperiodic and periodic tasks. Our framework provides configurable *admission controller* (AC), *load balancer* (LB), and *idle resetter* (IR) components which interact with application components and *task effector* (TE) components. The AC component provides online admission control and schedulability tests for tasks that arrive dynamically at runtime. The LB component provides an acceptable task assignment plan to the admission controller if the new arrival task is admissible. Each IR component reports all completed subjobs on one processor to the AC component when the processor becomes idle, so the AC component can remove their contributions from the calculated synthetic utilization, to reduce the pessimism of the AUB analysis at runtime according to the idle resetting rule. On each processor a TE component notifies the AC component when new jobs arrive, and releases admitted jobs.

Fig. 1 illustrates our distributed middleware architecture. All processors are connected by the TAO ORB's federated Event Channel (EC) [14], indicated by EC/ORB in Fig. 1. Black arrows in Fig. 1 represent an EC event being pushed or an ORB method call being sent. The EC pushes events through local event channels, gateways, and remote event channels to the events' consumers sitting on different processors. We deploy one AC component and one LB component which cooperate to perform task management on one processor, and one IR component and one TE component on each of multiple application processors.

Fig. 1 shows an example end-to-end task T_i composed of three consecutive subtasks, $T_{i,1}, T_{i,2}$, and $T_{i,3}$, executing on separate processors. $T_{i,1}$ and $T_{i,2}$ have duplicates on other application processors. An original component and its duplicate(s) are alternative application components that can execute the same subtask, with the actual subtask assignment decided by the LB component at runtime. For sake of discussion, assume that task T_i arrives at application processor 3. The TE component on that processor pushes a "Task Arrival" event to the AC component and holds the task until it receives an "Accept" event 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 lines and the dashed lines 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 a *task reallocation*.

An advantage of this centralized AC/LB architecture is that it does not require synchronization among distributed admission controllers. In contrast, in a distributed task

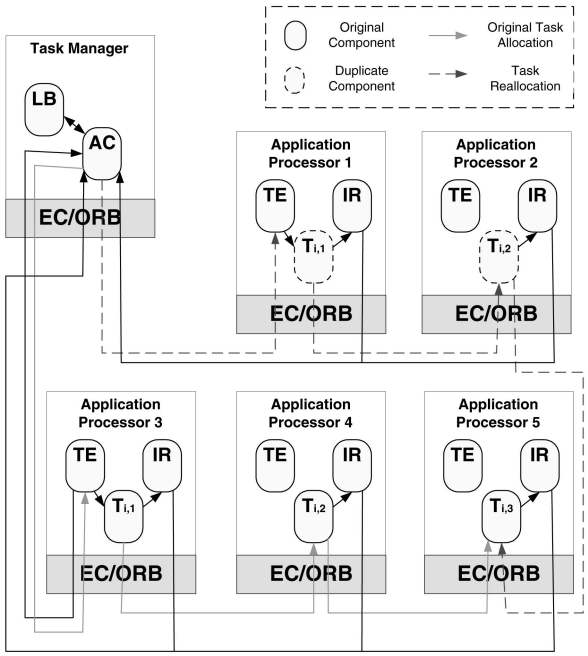

Fig. 1. Component Middleware Architecture: black arrows represent an event push or method call; original and duplicate components are alternatives for executing the same subtask; assume task T_i arrives at application processor 3.

management 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 multiple affected processors.

A potential disadvantage of the centralized architecture is that the AC component may become a bottleneck and, thus, affect scalability. However, the computation time of the schedulability analysis is significantly lower than task execution times in many DRS, which alleviates the scalability limitations of a centralized solution [5].

Centralized task management also could become a single point of failure, negatively impacting system availability and survivability. Admission control and load balancing could be replicated using existing active and passive fault-tolerance techniques for real-time systems [15], [16]. However, addressing a complete set of fault-tolerant task management issues is beyond the scope of this paper and is left as a potential future extension of this work. In summary, while our real-time component middleware approach can be extended to use a more distributed task management architecture, we have adopted a centralized approach with less complexity and overhead, which allows us to focus on achieving system flexibility through component middleware services.

4 Mapping DRS Characteristics to Middleware Strategies

A key contribution of this paper is categorizing characteristics that are common to a reasonably representative set of DRS applications, and mapping them to suitable middleware service strategies. In this section, we present a set of criteria used to categorize DRS characteristics, and analyze how to map those criteria to different service strategies supported by our middleware.

TABLE 1
Criteria and Middleware Service Strategies

	No	Yes	Some
C1:Job Skipping	AC per Task	AC per Job	
C2:Overhead Tolerance	No IR	IR per Job	IR per Task
C3:State Persistency	LB per Job	LB per Task	
C4:Component Replication	No LB	LB	

4.1 DRS Characteristics

We use four criteria to distinguish how DRS with aperiodic and periodic tasks can be supported: Job Skipping (criterion **C1**); Overhead Tolerance (criterion **C2**); State Persistency (criterion **C3**); and Component Replication (criterion **C4**).

Job Skipping means that some jobs of a task are executed while other jobs of the same task may not be admitted. Some applications, such as video streaming, and other loss-tolerant forms of sensing can tolerate job skipping, while in critical control applications, once a task is admitted, all its jobs should be allowed to execute.

Overhead Tolerance depends on an application's specific overhead constraints: we characterize different sources of overhead for our services in Section 7.3 so that developers of each application can decide whether those overheads would be excessive or acceptable if traded for improved schedulability.

State Persistency means that states are required to be preserved between jobs of a same task. For proportional control systems [17], task are stateless and only require current information, so jobs can be reallocated dynamically. However, for integral control systems [17], tasks require incremental calculation and are not suitable for job reallocation.

Component Replication depends on an application's throughput requirements. Replication is used here to reduce latency through load distribution, not for fault-tolerance purposes. Only those applications with replicated components can support task reallocation, whereas those that cannot be replicated (e.g., due to constraints on the locality of sensors or actuators) cannot support task reallocation.

According to these different application criteria, 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. Table 1 shows how these criteria help to classify DRS applications, which in turn allows selection of corresponding middleware service strategies. We have designed all strategies with corresponding configurable attributes, and provide a configuration preparser and a component configuration interface (described in Section 6) to allow developers to select and configure each service flexibly, according to each application's specific needs. We now examine the different strategies for each component and the trade-offs among them.

4.2 Admission Control (AC) Strategies

Admission control offers significant advantages for 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 the admission test only when a task first arrives while **AC-per-Job** performs the admission test whenever a job of the task arrives. Only applications satisfying criterion C1 are suitable for the second strategy, since it may not admit some jobs. Moreover, the second strategy reduces pessimism at the cost of increasing overhead. The application developer, thus, needs to consider trade-offs between overhead and pessimism in choosing a proper configuration.

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 [11], 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 subjob 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 result in pessimistic admission decisions [11].

AC-per-Job. If it is possible to skip a job of a periodic task (criterion C1), 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 is shown in Section 7.3 by our overhead measurements.

4.3 Idle Resetting (IR) Strategies

Without the AUB resetting rule, a job remains in the current set even if it has been completed, as long as its deadline has not expired. Therefore, the use of the resetting rule can remove the contribution of completed subjobs earlier than the deadline, which reduces the pessimism of the AUB schedulability test [5], [11]. There are three strategies to configure IR components in our approach, according to an application's overhead tolerance (criterion C2). 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 subjobs more frequently than the other two strategies. Although it has the least pessimism, it introduces the most overhead. The second strategy offers a trade-off between the first and the third strategies.

No IR. The first strategy is to use no resetting at all, so that if the subjobs complete their executions, the contributions of completed subjobs to the processor's synthetic utilization are not removed until the job 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 subjobs on one processor. Whenever the processor is idle, a lowest priority thread called an *idle detector* begins to run, and reports the completed *aperiodic* subjobs to the AC component through an "Idle Resetting" event. To avoid reporting repeatedly, the idle detector only reports when there is a newly completed aperiodic subjob whose deadline has not expired.

IR-per-Job. The third strategy is that each IR component records and reports not only the completed aperiodic subjobs but also the completed subjobs of *periodic* subtasks.

4.4 Load Balancing (LB) Strategies

Under AUB-based AC, load balancing can effectively improve system performance in the face of dynamic task arrivals [11]. We use a heuristic algorithm to assign subtasks to processors at runtime, which always assigns a subtask to the processor with the lowest synthetic utilization among all processors on which the application component corresponding to the task has been replicated (criterion C4).[1] Since migrating a subtask between processors introduces extra overhead, when we accept a new task, we only determine the assignment of that new task and do not change the assignment plan for any other task in the current task set. This service also has three strategies. The first strategy is suitable for applications which cannot satisfy criterion C4. The second strategy is most applicable for applications which satisfy both C4 and C3. The third strategy is most suitable for applications which only satisfy C4, but cannot satisfy criterion C3.

No LB. This strategy does not perform load balancing. Each subtask does not have a replica and is assigned to a 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. All jobs from a periodic task are allowed to be assigned to different processors when they arrive.

4.5 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 subjobs 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 that 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. With this degree of complexity in making correct configuration design decisions, an application developer would benefit from cognitive support in configuring the different strategies. An

1. The focus here is not on the load balancing algorithms themselves. Our configurable middleware may be easily extended to incorporate LB components implementing other load balancing algorithms according to each application's needs.

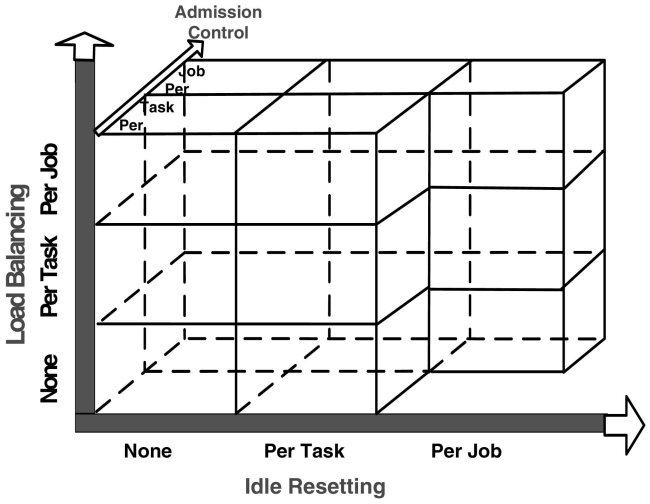

Fig. 2. Strategy dimensions of middleware services.

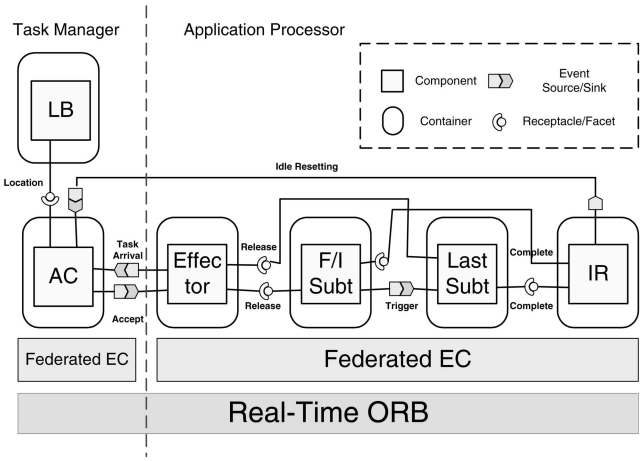

Fig. 3. Component implementation.

advantage of our middleware architecture and configuration engine is that they allow application developers to configure middleware services to achieve any valid combination of strategies, while disallowing invalid combinations, up front as we discuss in Section 6.

As Fig. 2 shows, the configuration choices can be divided into axes of strategy configurability for each of the three middleware services: admission control, idle resetting, and load balancing. Different configuration options in each of these axes and the impact they may have, as well as conflicting configurations, are thus delineated thoroughly and as we discuss in Section 6, form the basis for automated support of application developers in configuring the services our middleware provides.

5 Component Implementation

Configurable component middleware standards, such as the CCM [18], can help to reduce the complexity of developing DRS by defining a component-based programming paradigm. They also help by defining a standard configuration framework for packaging and deploying reusable software components. The CIAO [19] is an implementation of the Light Weight CCM specification [7] that is suitable for DRS. To support the different service strategies described in Section 4 and to allow flexible configuration of suitable combinations of those strategies for a variety of applications, we have implemented admission control, idle resetting, and load balancing services in 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 runtime through appropriate ports to form a DRS.

As Fig. 3 illustrates, we have designed and implemented six configurable components to support distributed real-time aperiodic and periodic end-to-end tasks using ACE/TAO/CIAO version 5.6/1.6/0.6. The dashed vertical line in Fig. 3 reflects the logical partitioning of task management and application processing components into separate processes. In our implementation, the task manager could run on an application processor, or on a separate processor as shown in Fig. 1 in Section 3. For efficiency, local interactions are implemented via method calls, while for flexibility, remote interactions are implemented via federated event handling.

The TE component holds the arriving tasks, waits for the AC component decision, and releases tasks. The Admission Control (AC) component decides whether to accept tasks. The Load Balancing (LB) component decides task allocations so as to balance the processors' synthetic utilizations. The First/Intermediate (F/I) Subtask component executes the first or an intermediate subtask at a given priority. The Last Subtask component executes the last subtask at a given priority. The Idle Resetting (IR) component reports the completed subjobs when a processor goes idle.

Each component may have several configurable attributes, so that it can be instantiated with a different criticality and execution time (for application components) or a different strategy (for AC, IR, and LB components). As we discussed in Section 3, our admission control and load balancing approaches adopt a centralized architecture, which employs one AC component instance and one LB component instance running on a central processor (called the "Task Manager" processor).

Each application processor contains one instance of a TE component and one instance of an IR component. The TE component on each processor reports the arrival of tasks on that processor to the AC component, which then releases or rejects the tasks based on the admission control policy. Each end-to-end task is implemented by a chain of F/I Subtask components and one Last Subtask component. We now describe the behavior of each component in detail.

TE component. When a task arrives, the TE component puts it into a waiting queue and pushes a "Task Arrival" event to the AC component. When the TE component receives an "Accept" event from the AC component, the corresponding task waiting in the queue will be released immediately. The TE component has two configurable attributes. One is a processor ID, which distinguishes TE component instances deployed on different processors. The other is the Per-job/Per-task attribute, which indicates

whether before releasing any job of a periodic task the TE component will hold it until receiving an "Accept" event from the AC component. If the attribute is set to be Per task, when a periodic task is admitted all subsequent jobs from that periodic task can be released immediately. These attributes can be set at the creation of a TE component instance and also may be modified at runtime.

First/Intermediate (F/I) and Last Subtask Components. Both the F/I and Last Subtask components execute application subtasks. The only difference between these two kinds of components is that the F/I 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. Each instance of these kinds of components contains a dispatching thread that executes a particular subtask at a specified priority. Both kinds of components have three configurable attributes. The first two attributes are the task 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 subjobs 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 subjobs complete. Both F/I Subtask and Last Subtask components call the "Complete" method of the local IR component instance when needed. The dispatching threads in an F/I Subtask or a Last Subtask component are triggered by either a "Release" method call from the local TE component instance or a "Trigger" event from a previous F/I Subtask component instance.

Idle Resetting (IR) Component. It receives "Complete" method calls from local F/I or Last Subtask components, and pushes "Idle Resetting" events to the AC component. It has one attribute, the processor ID, which distinguishes component instances sitting on different processors.

Admission Control (AC) Component. It consumes "Task Arrival" events from the TE components and "Idle Resetting" events from the IR components. It publishes "Accept" events to the TE components to allow task releases. It makes "Location" method calls on the LB component to ask for proposed task assignment plans. The AC component has a No LB/LB-per-task/LB-per-job attribute, which indicates whether load balancing is enabled, and if it is enabled whether it is per task or per job. If that attribute is set to LB-per-task, 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 LB-per-job means the subtask assignment plan can be changed for each job of an accepted periodic task.

Load Balancing (LB) Component. It receives "Location" method calls from the AC component, which fetches assignment plans for particular tasks. The LB component tries to balance the synthetic utilization among all processors, and may modify a previous allocation plan when a new job of the task arrives. It returns an assignment plan that is acceptable and attempts to minimize differences among 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.

6 Deployment and Configuration

While our configurable components represent an important step toward flexible middleware services for handling aperiodic and periodic events, DRS developers still face the challenges of choosing the best combinations of strategies and assembling and deploying the components, which are tedious and error prone if performed by hand. Therefore, we have developed a tool that automates the selection, deployment, and configuration of these components. Our tool has two key advantages: 1) it allows application developers to specify the characteristics of the DRS and automatically map them to suitable middleware strategies, and 2) it identifies incorrect combinations of service strategies to prevent erroneous middleware configurations. CIAO's realization of the OMG's Light Weight Deployment and Configuration specification [7] is called the Deployment and Configuration Engine (DAnCE) [9]. 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 DRS 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.

Front-end Configuration Engine. Although tools such as CoSMIC [20] can help generate the XML files, those tools do not consider the configuration requirements of the new services we have created. We, therefore, provide a specific configuration engine (illustrated in Fig. 4) that acts as a front end to DAnCE, to configure our services for application developers who require configurable aperiodic scheduling support. This extension to DAnCE helps to alleviate complexities associated with deploying and configuring our services. The application developer first provides a workload specification file which describes each end-to-end task and where its subtasks execute. Our front-end configuration engine asks the application developer to specify the characteristics of the DRS, via a simple textual interface as shown in Fig. 5.

The front-end configuration engine parses the workload specification file and automatically maps application characteristics specified by the developer to proper configuration settings for the admission control, idle resetting, and load balancing services. Finally, an XML-based deployment plan is generated, which can be recognized by DAnCE. As an example, Fig. 4 shows one set of answers to those four questions. Based on those answers, the AC, IR, and LB services should all be configured using per-task (PT) strategy. Fig. 4 also shows part of the XML file generated by our configuration engine, with the LB strategy setting of

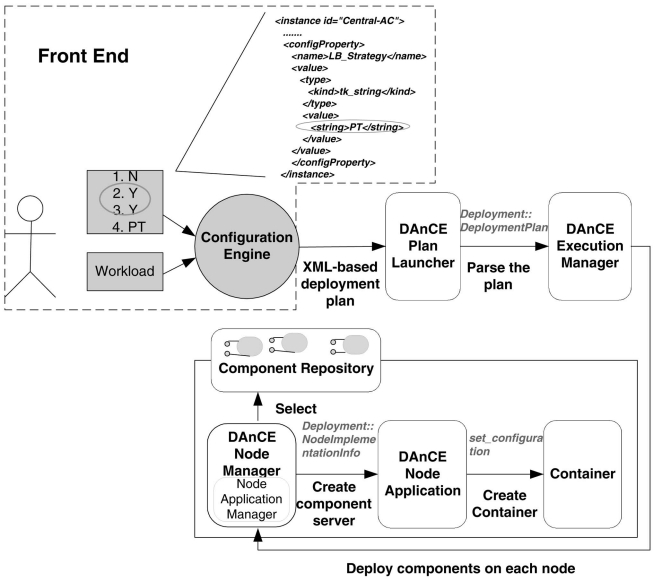

Fig. 4. Front-end engine and its interaction with DAnCE.

PT, which is due to the developer's answers to second and third questions.

To enforce end-to-end deadline monotonic scheduling, the F/I Subtask and Last Subtask components both expose an attribute called "priority." When our configuration engine reads the workload specification 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 configuration settings, 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 4.5. Since a developer might specify incompatible service configuration combinations, our approach should be able to detect and disallow them. If application characteristics are not provided by the developers, our configuration engine also can supply default configuration settings, i.e., per-task admission control, idle resetting, and load balancing.

We have used the <configproperty> feature of DAnCE to extend the set of attributes that can be 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. 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

```
(1) Does your application allow job skipping?
    [yes (Y), no (N)]
(2) Does your application have replicated components?
    [yes (Y), no (N)]
(3) Does your application require state persistence?
    [yes (Y), no (N)]
(4) How much extra overhead can you accept  as it
    potentially improves schedulability?
    [none (N), some per task (PT), some per job (PJ)]
```

Fig. 5. Questions to determine characteristics for strategy selection.

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.

7 Experimental Evaluations

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 100 Mbps Ethernet switch. Two are Pentium-IV 2.5 GHz machines with 1G RAM and 512,000 cache each, two are Pentium-IV 2.8 GHz machines with 1G RAM and 512,000 cache each, and the other two are Pentium-IV 3.4 GHz machines with 2G RAM and 2,048,000 cache each. Each machine runs version 2.4.22 of the KURT-Linux operating system. One Pentium-IV 2.5 GHz machine is used as a central task manager where the AC and LB components are deployed. The other five machines are used as application processors on which TE, F/I Subtask, Last Subtask, and IR components are deployed.

7.1 Random Workloads

We first randomly generated 10 sets of nine tasks, each including four aperiodic tasks and five periodic tasks. The number of subtasks per task is uniformly distributed between 1 and 5. Subtasks are randomly assigned to five 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 four application processors.

In this experiment, we evaluated all 15 reasonable combinations of strategies, since it is convenient to choose and run different combinations with the help of our configuration engine. We ran 10 task sets using each combination and compared them. Each task set ran for 5 minutes for each combination. The performance metric we used in these evaluations is the *accepted utilization ratio*, i.e., the total utilization of jobs actually released divided by the total utilization of all jobs arriving. To be concise, we use capital letters to represent strategies: **N** when a service is **not** enabled in this configuration; **T** when a service is enabled for each **task**; and **J** when a service is enabled for

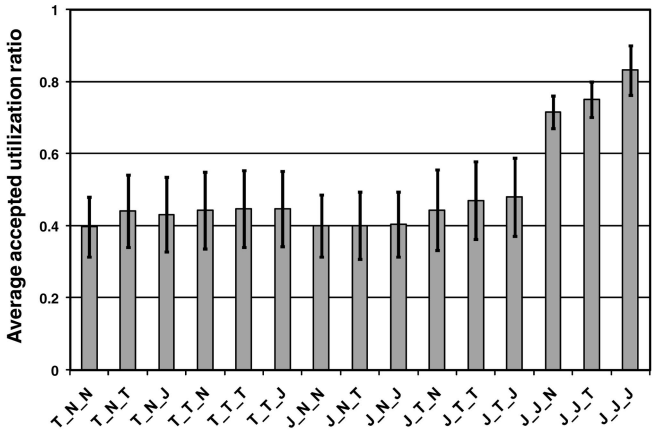

Fig. 6. Accepted utilization ratio for different (AC, IR, and LB) combinations.

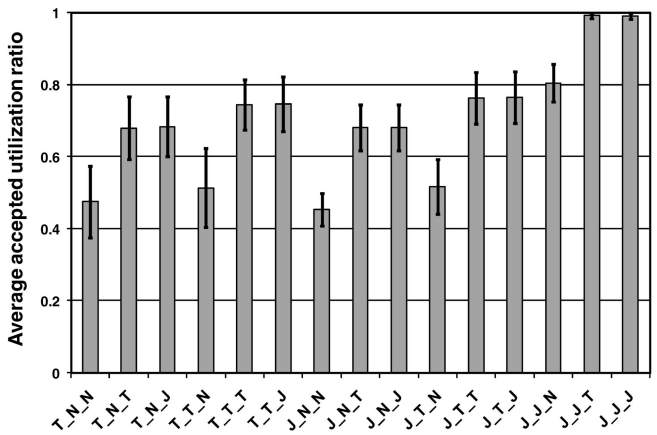

Fig. 7. LB strategy comparison for different (AC, IR, and LB) combinations.

each **job** of a task. In the following figures, a three element tuple denotes each combination of settings for the three configurable services: first for the admission control service, then for the idle resetting service, and last for the load balancing service.

The bars in Fig. 6 show the average results over the 10 task sets. As is shown in Fig. 6, enabling either idle resetting or load balancing can increase the utilization of tasks admitted. Moreover, the experiment shows that enabling IR-per-job (*_J_*) significantly outperforms the configurations which enable IR-per-task (*_T_*) or not at all (*_N_*). This is because IR-per-job removes the contribution of all completed periodic subjobs to the synthetic utilizations which greatly helps to admit more jobs. Enabling all three services per job (J_J_J) performed comparably to the other (J_J_*) configurations (averaging higher though the differences were not significant) and outperformed all other configurations significantly, even though the J_J_J configuration introduces the most overhead. We also notice the difference is small when we only change the configuration of the LB component and keep the configuration of other two services the same. This is because when we randomly generated these 10 task sets, the resulting synthetic utilization of each processor was similar. To examine the potential benefit of the LB component, we designed another experiment that is described in the next section.

7.2 Imbalanced Workloads

In the second experiment, we use an imbalanced workload. It is representative of a dynamic DRS in which a subset of the system processors may experience heavy load. For example, in an industrial control system, a blockage in a fluid flow valve may cause a sharp increase in the load on the processors immediately connected to it, as aperiodic alert and diagnostic tasks are launched. In this experiment, we divided the five application processors into two groups. One group contains three processors hosting all tasks. The other group contains two processors hosting all duplicates. Ten task sets are randomly generated as in the above experiment, except that all subtasks were randomly assigned to three 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 replica sitting on one processor in the second group.

Each of 10 task sets was run for the 15 different valid strategy setting combinations, and for each combination we then averaged the accepted utilization ratio over the 10 results. We varied the load balancing strategy from none to per task, then to per job, for each of the five valid combinations of admission control and idle resetting strategies. As Fig. 7 shows, under the conditions our experiment studied LB-per-Task provides a significant improvement when compared with the results without LB. However, there is not much difference between LB-per-Task versus LB-per-Job. Note that since there are five application processors and a total task utilization of 2.1, if we can assign tasks almost evenly among processors through load balancing all tasks are schedulable (as indicated by the accepted utilizations near 1.0 in Fig. 7 for J_J_T and J_J_J). However, two processors in the second group are not used when load balancing is disabled, resulting in a lower accepted utilization ratio with J_J_N.

From the two experiments described in this section and in Section 7.1, we found that configuring different strategies according to application characteristics can have a significant impact on the performance of a DRS with aperiodic and periodic events. Our design of the AC, IR, and LB services as easily configurable components allows application developers to explore and select valid configurations based on the characteristics and requirements of their applications, and based on the trade-offs indicated by these empirical results.

7.3 Overheads of Service Components

To evaluate the efficiency of our component-based middleware services, we measured overheads using three of the processors to run application components and another processor to run the AC and LB components. The workload is randomly generated in the same way as described in Section 7.1, 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 TE

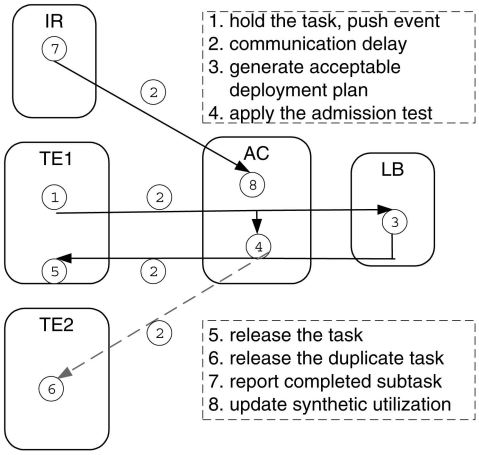

Fig. 8. Sources of overhead/delay.

component TE1, after which AC and LB components run the task in component TE1 or reallocate it to another TE component, TE2. Fig. 8 shows how the total delay for each service includes the costs of operations located in several components. Table 2 lists the operation numbers shown in Fig. 8 to provide a detailed accounting of the delays resulting from different combinations of service configurations.

To calculate the delays for AC without LB, AC with LB without reallocation, and LB without reallocation, 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 LB component reallocates 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 our experiment environment does not provide sufficiently high-resolution time synchronization among processors, which is an inherent limitation for many DRS. We, therefore, measure the overheads on all involved processors individually, then add them together plus twice the communication delays (step 2 in Fig. 8) between the processors. Three processors are involved: the processor where the task arrives (step 1), the central task manager processor (step 3), and the processor where the duplicate task is released (step 6). We ran this experiment using KURT-Linux version 2.4.22, which provides a CPU-supported time stamp 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 and the admission control processor 1,000 times, then calculated the mean and max value among 1,000 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 LB 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 IR component, we divide its execution into two parts. The small overhead on the admission control component must be counted in the overall delay. 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 component, it does not affect performance, which is why we report the two parts separately in Table 2. From the results in Table 2, we can see that all of the delays induced by our configurable services are less than 2 ms, which is acceptable for many DRS. For applications with tight schedules, a developer can make further decisions on how to configure services based on this delay information and based on the effects of the different configurations on task management, which we discussed in Section 4.

TABLE 2
Service Overheads (μs)

	mean	max
AC without LB (1+2+4+2+5)	1114	1248
AC with LB (1+2+3+2+5) (no re-allocation)	1116	1253
AC with LB (1+2+3+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	1319
IR (on AC side) (8)	17	18
IR (other part) (7+2)	662	683
Communication Delay (2)	322	361

8 Related Work

In this section, we consider related work on middleware designed for managing applications QoS requirements, of which real-time requirements are a subset. We first describe approaches that are not based on component middleware, and then consider component-based approaches.

QoS-aware middleware. Quality Objects (QuO) [21], [22] is an adaptive middleware framework developed by BBN Technologies that allows developers to use aspect-oriented software development techniques to separate the concerns of QoS programming from application logic. A *Qosket* is the unit of encapsulation and reuse in QuO. QuO emphasizes dynamic QoS provisioning whereas our approach emphasizes static QoS provisioning. The dynamicTAO [23] project applies reflective techniques to reconfigure ORB policies and mechanisms at runtime. Similar to dynamicTAO, the Open ORB [24] project also aims at highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements. Zhang and Jacobsen [25] also use aspect-oriented techniques to improve the customizability of the middleware core infrastructure at the ORB level.

QoS-aware component middleware. Component middleware architectures have been leveraged to enable metaprogramming of QoS attributes in a number of ways. For example, aspect-oriented techniques can be used to plug in different behaviors [26] into the *containers* that host components. This approach is similar to ours in that it provides mechanisms to configure system attributes at the middleware level. de Miguel [27] further develops QoS-enabled containers by extending an EJB container interface to allow

the exchange of QoS-related information among component instances. To take advantage of this *QoS-container*, a component must implement QoSBean and QoSNegotiation interfaces. However, this requirement increases dependence among component implementations. The QoS-Enabled Distributed Objects (Qedo) [28] project is another effort to make QoS support an integral part of the CCM. Qedo's extensions to the CCM container interface and Component Implementation Framework (CIF) require component implementations to interact with the container QoS interface and negotiate the level of QoS contract directly. Although this approach is suitable for certain applications, it tightly couples the QoS provisioning and adaptation behaviors into the component implementation, which may limit the reusability of components. In comparison, our approach explicitly avoids this coupling and composes real-time attributes declaratively. There have been several other efforts to introduce QoS attributes in conventional component middleware platforms. The FIRST Scheduling Framework (FSF) [29] proposes to compose several applications and to schedule the available resources flexibly while guaranteeing hard real-time requirements. A real-time component type model [30], which integrates QoS facilities into component containers also was introduced based on the EJB and RMI specifications. A schedulability analysis algorithm [31] 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 aperiodic and periodic end-to-end tasks offered by our approach.

9 Conclusions

The work presented in this paper represents a promising step toward configurable middleware services for diverse DRS applications with aperiodic and periodic events. We have identified a common set of key characteristics representative of many DRS applications, and have shown how to map those characteristics to suitable strategies for real-time middleware task management services. We have designed and implemented configurable middleware components that provide effective online admission control and load balancing and can be easily configured and deployed on distributed computing platforms. The front-end configuration engine we have developed can automatically process specified application characteristics to generate a corresponding deployment plan for DAnCE, thus, making it easier for developers to select suitable configurations, and to avoid invalid ones. Results of the experiments we have conducted to evaluate our approach show that 1) our configurable component middleware approach is well suited for supporting different applications with alternative characteristics and requirements, and 2) the delays imposed by our component middleware services are below 2 ms on a representative Linux platform.

The purpose of this research is to demonstrate the efficacy of allowing a variety of strategy combinations to be configured, to support applications with different criteria. While application-specific studies would certainly offer further insight into the trade-offs among strategy configurations in each application domain, the random task sets used in this paper demonstrate the potential benefit of having such flexibility. The results presented in Section 7 encourage further investigation as future work into how well specific task sets from real-time application domains such as real-time image transmission [32], shipboard computing [11], and avionics mission computing [33] can be supported using the guidance offered in Section 4.1.

Acknowledgments

This research has been supported in part by the US National Science Foundation (NSF) grant CCF-0615341 (EHS) and the NSF CAREER award CNS-0448554. Y. Zhang was with the Department of Computer Science and Engineering, Washington University, Campus Box 1045, One Brookings Drive, St. Louis, MO 63130-4899.

References

[1] D.C. Schmidt, "Successful Project Deployments of ACE and TAO," www.cs.wustl.edu/~schmidt/TAO-users.html, Washington Univ., 2009.
[2] *Real-Time CORBA Specification,* 1.1 ed., Object Management Group, Aug. 2002.
[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java.* Addison-Wesley, 2000.
[4] G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems That Allow Skips," *Proc. IEEE Real-Time Systems Symp. (RTSS),* 1995.
[5] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker, "Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* 2007.
[6] Institute for Software Integrated Systems, "Component Integrated ACE ORB (CIAO),"www.dre.vanderbilt.edu/CIAO/, Vanderbilt Univ., 2009.
[7] *Light Weight CORBA Component Model Revised Submission,* OMG Document Realtime/03-05-05 ed., Object Management Group, May 2003.
[8] Institute for Software Integrated Systems, "The ACE ORB (TAO),"www.dre.vanderbilt.edu/TAO/, Vanderbilt Univ., 2009.
[9] G. Deng, D.C. Schmidt, C. Gill, and N. Wang, *QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems.* CRC Press, 2007.
[10] L. Sha et al., "Real Time Scheduling Theory: A Historical Perspective," *J. Real-Time Systems,* vol. 10, pp. 101-155, 2004.
[11] T.F. Abdelzaher, G. Thaker, and P. Lardieri, "A Feasible Region for Meeting Aperiodic End-to-End Deadlines in Resource Pipelines," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* 2004.
[12] B. Andersson and C. Ekelin, "Exact Admission-Control for Integrated Aperiodic and Periodic Tasks," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* 2005.
[13] J. Strosnider, J.P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Real-Time Environments," *IEEE Trans. Computers,* vol. 44, no. 1, Jan. 1995.
[14] T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Application (OOPSLA),* 1997.
[15] P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA," *Concurrency and Computation: Practice and Experience,* vol. 17, pp. 1527-1545, 2005.
[16] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D.C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* 2009.
[17] F.H. Raven, *Automatic Control Engineering,* fifth ed. Mcgraw-Hill, 1994.
[18] *CORBA Components,* OMG Document Formal/2002-06-65 ed., Object Management Group, June 2002.

[19] N. Wang, C. Gill, D.C. Schmidt, and V. Subramonian, "Configuring Real-Time Aspects in Component Middleware," *Proc. Int'l Symp. Distributed Objects and Applications (DOA),* 2004.
[20] A. Gokhale, "Component Synthesis Using Model Integrated Computing," www.dre.vanderbilt.edu/cosmic, 2003.
[21] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging Quality of Service Control Behaviors for Reuse," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC),* 2002.
[22] J.A. Zinky, D.E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems,* vol. 3, no. 1, pp. 1-20, 1997.
[23] F. Kon, F. Costa, G. Blair, and R.H. Campbell, "The Case for Reflective Middleware," *Comm. ACM,* vol. 45, no. 6, pp. 33-38, June 2002.
[24] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB 2," *IEEE Distributed Systems Online,* vol. 2, no. 6, June 2001.
[25] C. Zhang and H.-A. Jacobsen, "Resolving Feature Convolution in Middleware Systems," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Application (OOPSLA),* 2004.
[26] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers," *Proc. Int'l Workshop Component-Oriented Programming (WCOP),* 2001.
[27] M.A. de Miguel, "QoS-Aware Component Frameworks," *Proc. Int'l Workshop Quality of Service (IWQoS),* 2002.
[28] FOKUS, "Qedo Project Homepage,"http://qedo.berlios.de/, 2009.
[29] 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," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* 2006.
[30] M.A. de Miguel, "Integration of QoS Facilities into Component Container Architectures," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC),* 2002.
[31] J.L. Lorente, G. Lipari, and E. Bini, "A Hierarchical Scheduling Model for Component-Based Real-Time Systems," *Proc. Workshop Parallel and Distributed Real-Time Systems (WPDRTS),* 2006.
[32] X. Wang, M. Chen, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill, "Control-Based Adaptive Middleware for Real-Time Image Transmission over Bandwidth-Constrained Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 6 pp. 779-793, June 2008.
[33] C. Gill, F. Kuhns, D.C. Schmidt, and R. Cytron, "Empirical Differences between COTS Middleware Scheduling Paradigms," *Proc. Int'l Symp. Distributed Objects and Applications (DOA '02),* Oct. 2002.

Yuanfang Zhang received the BS and MS degrees in computer science from Fudan University, China, in 1999 and 2002, respectively. She received the PhD degree in computer science from Washington University in St. Louis in 2008. Her research interests include real-time middleware, real-time systems, and multicore platforms. She is now with the Cloud Computing Future team at Microsoft Research, Redmond, Washington.

Christopher D. Gill is an associate professor of computer science and engineering in the Department of Computer Science and Engineering, Washington University in St. Louis. His research interests include formal modeling, verification, implementation, and empirical evaluation of policies and mechanisms for enforcing timing, concurrency, footprint, fault tolerance, and security properties in distributed, mobile, embedded, and real-time systems. He developed the Kokyu real-time scheduling and dispatching framework that has been used in several AFRL and US Defense Advanced Research Projects Agency (DARPA) projects. He led the development of the nORB small-footprint real-time object request broker at Washington University in St. Louis. He has also led research projects under which a number of real-time and fault-tolerant services for The ACE ORB (TAO) and the Component-Integrated ACE ORB (CIAO) were developed. He has more than 50 refereed and invited technical publications and has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed real-time and embedded computing. He is a member of the IEEE and the IEEE Computer Society.

Chenyang Lu received the BS degree from the University of Science and Technology of China in 1995, the MS degree from the Chinese Academy of Sciences in 1997, and the PhD degree from the University of Virginia in 2001, in computer science. He is an associate professor of computer science and engineering at Washington University in St. Louis. He is the author and coauthor of more than 80 publications, and received the NSF CAREER Award in 2005 and the Best Paper Award at the International Conference on Distributed Computing in Sensor Systems in 2006. He is an associate editor of the *ACM Transactions on Sensor Networks* and the *International Journal of Sensor Networks*, and guest editor of the Special Issue on Real-Time Wireless Sensor Networks of Real-Time Systems. He also served as program chair and general chair of the IEEE Real-Time and Embedded Technology and Applications Symposium in 2008 and 2009, track chair on Wireless Sensor Networks for IEEE Real-Time Systems Symposium in 2007 and 2009, and demo chair of the ACM Conference on Embedded Networked Sensor Systems in 2005. He serves on the executive committee of the IEEE Technical Committee on Real-Time Systems. His research interests include real-time embedded systems, wireless sensor networks, and cyber-physical systems. He is a member of the ACM and the IEEE.