Report Number: WUCSE-2007-47

2007

Roadmap Analysis of Protein-Protein Interactions. Master's Thesis, August 2007

Brian C. Haynes

The ability to effectively model the interaction between proteins is an important and open problem. In molecular biology it is well accepted that from sequence arises form and from form arises function but relating structure to function remains a challenge. The function of a given protein is defined by its interactions. Likewise a malfunction or a change in protein-protein interactions is a hallmark of many diseases. Many researchers are studying the mechanisms of protein-protein interactions and one of the overarching goals of the community is to predict whether two proteins will bind, and if so what the final conformation... **Read complete abstract on page 2.**

Roadmap Analysis of Protein-Protein Interactions. Master's Thesis, August 2007

Brian C. Haynes

Complete Abstract:

The ability to effectively model the interaction between proteins is an important and open problem. In molecular biology it is well accepted that from sequence arises form and from form arises function but relating structure to function remains a challenge. The function of a given protein is defined by its interactions. Likewise a malfunction or a change in protein-protein interactions is a hallmark of many diseases. Many researchers are studying the mechanisms of protein-protein interactions and one of the overarching goals of the community is to predict whether two proteins will bind, and if so what the final conformation will be. Attention is seldom paid to the association pathways that allow two proteins to bind. Evidence has shown that the information in the association pathways can play a vital role in understanding the interaction itself. This thesis presents a novel and scalable approach to computing association pathways between two proteins using the Probabilistic Roadmap (PRM) framework. We will discuss the challenges in extending PRM to the domain of protein-protein interactions such as performing structural mappings in a reduced space of flexibility, and sampling high dimensional conformation spaces. We will present analysis of individual association pathways as well as methods for estimating collective properties of the energy landscape. Our results indicate that these methods can discriminate between true and false protein binding interfaces. Finally, we will present condensing methods such as pathway clustering and visualization using dimensionality reduction that can be be applied to create compact representations of the interaction space.

2007-46

Customizing 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

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

Yuanfang Zhang, Christopher Gill and Chenyang Lu

Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{yfzhang, cdgill, lu}@cse.wustl.edu

Abstract. Many distributed real-time applications must handle mixed aperiodic and periodic 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 aperiodic and periodic 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 the effectiveness of, our configurable component middleware approach in supporting different applications with aperiodic and periodic tasks.

1 Introduction

Many distributed real-time systems must handle a mix of aperiodic and periodic 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, e.g., to put an industrial process into a fail-safe mode. User inputs and sensor readings may trigger other real-time aperiodic tasks as well.

However, there is a significant gap between the needs of these applications and the support provided to them even by state-of-the-art real-time middleware. Specifically, while significant theoretical results on aperiodic scheduling [1] have been achieved, there is a growing 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 have developed the first middleware-layer on-line admission control service supporting both aperiodic and periodic end-to-end tasks [2]. However, a more complete set of inter-operating services is needed, and for many of those services multiple strategies for their operation should be supported. For example, even for a single theoretic technique such as aperiodic utilization bound (AUB) [3], an admission control service may have alternative strategies, the effectiveness of which depends significantly on workload characteristics and application requirements. Our original admission control service had a single fixed strategy for when admissability was decided, which was only at the first arrival of each task. However, as we describe in Section 3.2,

for some applications deciding admissibility for each separate job of a periodic task may be a better alternative to reduce pessimism of the admission decision.

Moreover, it is essential to make those strategies easily and individually configurable, which is difficult for implementations that rely directly on distributed object middleware (like our original admission control service, which was based on TAO [4]). Specifically, in those implementations changing the supported strategy requires explicit changes to the service code itself, which can be tedious and error-prone in practice. Not only must appropriate services expose a variety of strategies, but the configuration of those strategies must be supported in a flexible yet principled way, so that system developers are able to explore alternative configurations but invalid configurations cannot be chosen by mistake. Providing configurable middleware service strategies to real-time applications with aperiodic and periodic tasks thus faces several important challenges:

- new and existing services for real-time systems with aperiodic and periodic tasks must provide configurable strategies, and configuration tools must be added or extended to allow configuration of those strategies;
- the specific criteria that distinguish which service strategies are preferable must be identified, and applications must be categorized according to those criteria;
- appropriate combinations of services' strategies must be identified for each such application category, according to its characteristic criteria;
- invalid combinations of strategies must be identified, and then disallowed during configuration; and
- empirical studies must be conducted to evaluate overheads and trade-offs among alternative valid combinations of service strategies.

To address these challenges, and thus to enhance support for diverse distributed real-time applications with aperiodic and periodic tasks, we have designed and implemented a new set of inter-operating services as configurable middleware components, atop the CIAO [5] quality-of-service (QoS)-aware component middleware platform. We have also developed a new front-end for the DAnCE [6] QoS-enabled component deployment and configuration engine, 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 example of configurable component middleware services supporting multiple admission control and load balancing strategies for aperiodic and periodic end-to-end real-time tasks; (2) developed a novel component configuration pre-parser and interfaces to select and configure real-time admission control and load balancing services flexibly at system deployment time; (3) defined categories of 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 tasks, 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 flexibility and applicability of distributed real-time middleware for systems with aperiodic and periodic tasks.

Section 2 introduces the services provided by the CIAO and DAnCE middleware frameworks which our work extends, and provides background information on a spe-

cific approach to supporting aperiodic tasks, whose features our work makes configurable. 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 extensions, which can flexibly configure different strategies for our services according to each application's requirements. Section 5 evaluates the performance of our approach including trade-offs among different service strategy combinations, and characterizes the overheads introduced by our approach. Section 6 presents a survey of other related work. Finally, we offer concluding remarks in Section 7.

2 Background

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

The Component-Integrated ACE ORB (CIAO) [5] implements the Lightweight CCM specification [7] (which in contrast to the full CCM is better suited to the needs of real-time and embedded systems) and is built atop the TAO [4] 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 real-time policies as installable and configurable units. However, CIAO does not support aperiodic task scheduling, admission control or load balancing. We base our approach on extending CIAO to configure and manage aperiodic task support.

DAnCE [6] is a QoS-enabled component deployment and configuration engine that implements the Object Management Group (OMG)'s Deployment and Configuration specification [8]. DAnCE parses component configuration/deployment descriptions and automatically configures ORBs, containers, and server resources at system initialization time, to enforce end-to-end QoS requirements. However, DAnCE does not provide essential features needed to configure our admission control and load balancing services correctly, e.g., to disallow invalid combinations of service strategies.

Aperiodic Task Support: Aperiodic tasks have been studied extensively in real-time scheduling theory, including work on aperiodic servers that integrate scheduling of aperiodic and periodic tasks [9–17]. New schedulability tests based on aperiodic utilization bounds [3] and a new admission control approach [18] also were introduced recently. In [2], we implemented and evaluated services for two suitable aperiodic scheduling techniques (aperiodic utilization bound [3] and deferrable server [9]). Since aperiodic utilization bound (AUB) has a comparable performance to deferrable server, and requires less complex scheduling mechanisms in middleware, we focus exclusively on the AUB scheduling technique in this paper. Our experiences with AUB reported in this

paper illustrate how configurability of other scheduling techniques can be integrated within real-time component middleware in a similar way.

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

In AUB [3], 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. 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 condition holds [3]:

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

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

Note that 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 [3]. When a processor becomes idle, the contribution of all completed subtasks to the processor's synthetic utilization can be removed without affecting the correctness of the schedulability condition (Condition 1). Since the resetting rule introduces extra overhead, it should be made configurable whether the contribution of only aperiodic subtasks or of both aperiodic and periodic subtasks can be removed early. Under AUB-based schedulability analysis, load balancing also can effectively improve system performance [3]. However some applications require persistent state preservation between jobs of the same task, so it should be made configurable whether a task can be reassigned to a different processor at each release.

3 Component Middleware Approach

In this section, we introduce our configurable service framework, and describe different application criteria and their relationships to different configurable strategies. We also discuss the implementation of the configurable components needed by our approach.

3.1 Architecture Overview

To support end-to-end aperiodic and periodic tasks, we have developed a new middleware architecture that 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 aperiodic and periodic tasks. Our framework provides configurable *admission controller* (AC), *idle resetter* (IR) and *load balancer* (LB) components which interact with application components through *task effector* (TE) components. The AC component provides on-line admission control and schedulability tests for tasks that arrive dynamically at run time. The LB component provides an acceptable task assignment plan to the admission controller if the new arrival task is admittable. The IR component reports all completed subtasks on one processor to the AC component when the processor becomes idle, so the AC component can remove their expected utilization to reduce the pessimism of the AUB analysis at run-time.

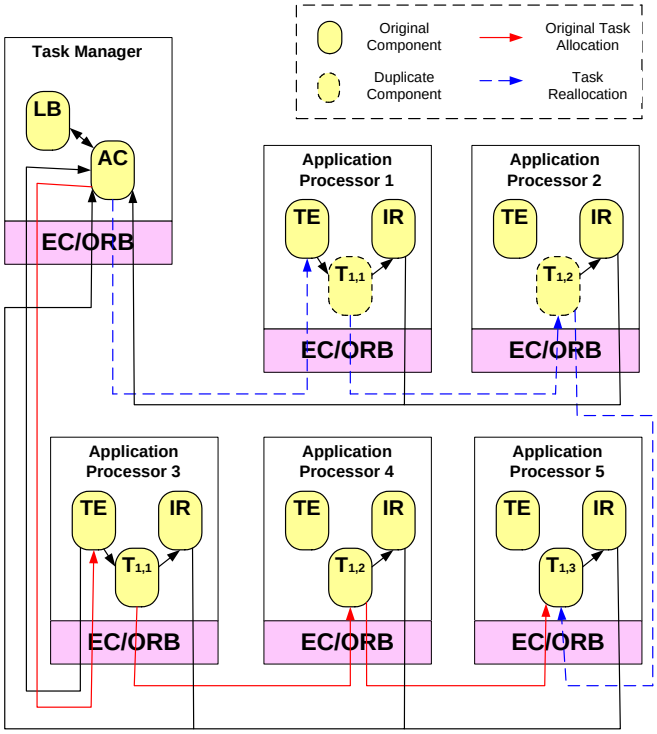

Fig. 1. Distributed Middleware Architecture

Figure 1 illustrates our distributed configurable component middleware architecture. All processors are connected by TAO's federated event channel [19] which 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 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 separate processors. $T_{i,1}$ and $T_{i,2}$ have duplicates on other application processors. When task T_i arrives at an application processor, the task effector component on that processor pushes a "Task Arrive" 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 a *task reallocation*.

A key advantage of this 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 multiple affected processors. A potential disadvantage of the centralized architecture is that the AC component may become a communication bottleneck and thus affect scalability. Therefore, our current middleware architecture is suitable for small to medium scale real-time systems in which processors are connected by high-speed real-time networks. However, the computation time of the schedulability analysis is significantly lower than task execution times in many high performance real-time applications, which may alleviate the scalability limitations of a centralized solution somewhat. In summary, while our real-time component middleware approach can be extended to use a more distributed architecture, we have focused first on a centralized approach with less complexity and overhead. As future work we plan to examine the benefits and costs of decentralized admission control and load balancing.

3.2 Applications Criteria and Middleware Strategies

Three criteria distinguish how different applications with aperiodic tasks should be supported: whether application components are replicated on multiple processors (criterion **C1**); whether persistent state must be preserved between jobs of a same task (criterion **C2**); and whether the application can tolerate job skipping (criterion **C3**). 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 and the application's overhead constraints. 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 with corresponding configurable component attributes, and provide a configuration pre-parser and a component configuration interface to allow developers to select and configure each 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.

Admission Control (AC) Strategies: Admission control offers significant advantages for systems with aperiodic and periodic tasks, by providing on-line 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 C3 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 inter-arrival 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 [3], 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 result in pessimistic admission decisions [3].

AC per Job: If it is possible to skip a job of a periodic task (criterion C3), 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 5.3 by our overhead measurements.

Idle Resetting (IR) Strategies: The use of a resetting rule can reduce the pessimism of the AUB schedulability test significantly [3, 2]. There are three ways to configure IR components in our approach. 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 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 subtasks complete their executions, the contributions of completed jobs to the processor's synthetic utilization are 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, and reports the completed *aperiodic* jobs 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 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.

Load Balancing (LB) Strategies: Under AUB-based AC, load balancing can effectively improve system performance in the face of dynamic task arrivals [3]. 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 on which the application component corresponding to the task has been replicated (criterion C1). [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 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.
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.

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. An 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.

3.3 Component Implementation

Configurable component middleware standards, such as the CORBA Component Model (CCM) [20], can help to reduce the complexity of developing 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) [21] is an implementation of the

[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.

Lightweight CCM specification [7] that is specifically designed and optimized for distributed real-time systems. To support the different strategies described in Section 3.2, 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 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 run-time through appropriate ports to form a distributed real-time application.

As Figure 2 illustrates, we have designed and implemented 6 configurable components to support distributed real-time aperiodic and periodic tasks end-to-end, using ACE/TAO/CIAO version 5.5.1/1.5.1/0.5.1. The Task Effector (TE) component holds

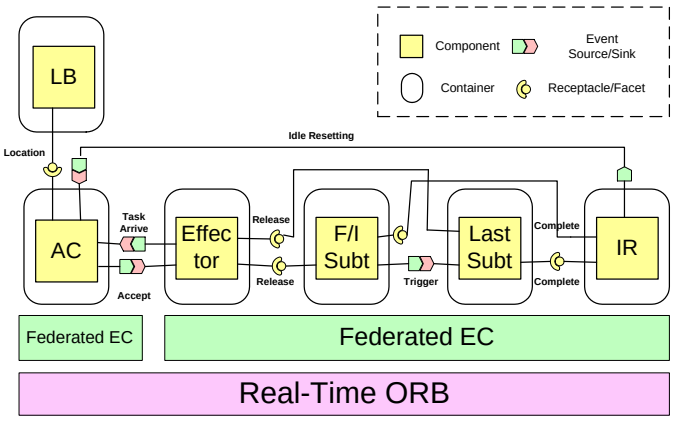

Fig. 2. Component Implementation

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 computes 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 records and reports the completed subtasks when a processor goes idle.

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). As we discussed in Section 3.1, 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 decision. The IR component on each application processor records and reports the completed subtasks on that processor to the AC component, whenever that processor goes idle. 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 kind of component in more detail.

Task Effector (TE) Component: When a task arrives, the TE component puts it into a waiting queue and pushes a "Task Arrive" 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 is used to distinguish TE 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 TE component will hold it until receiving an "Accept" event from the AC component. Otherwise, when a periodic task is admitted the AC component will reserve 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 a TE component instance and also may be modified at run-time.

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 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 F/I Subtask Component 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. Both F/I Subtask and Last Subtask components call the "Complete" method of the local IR component instance when a subtask completes.

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 is used to distinguish the component instances sitting on different processors.

Admission Control (AC) Component: It consumes "Task Arrive" 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 task.

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

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. CIAO's realization of the OMG's Deployment and Configuration specification [8] 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.

Front-end Configuration Engine: Although tools such as CoSMIC [22] 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 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 two text files. One is a configuration file shown in Figure 3, which consists of the configuration settings 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 part of the XML file generated by our configuration engine, with the LB strategy (LB_Strategy) setting of PT.

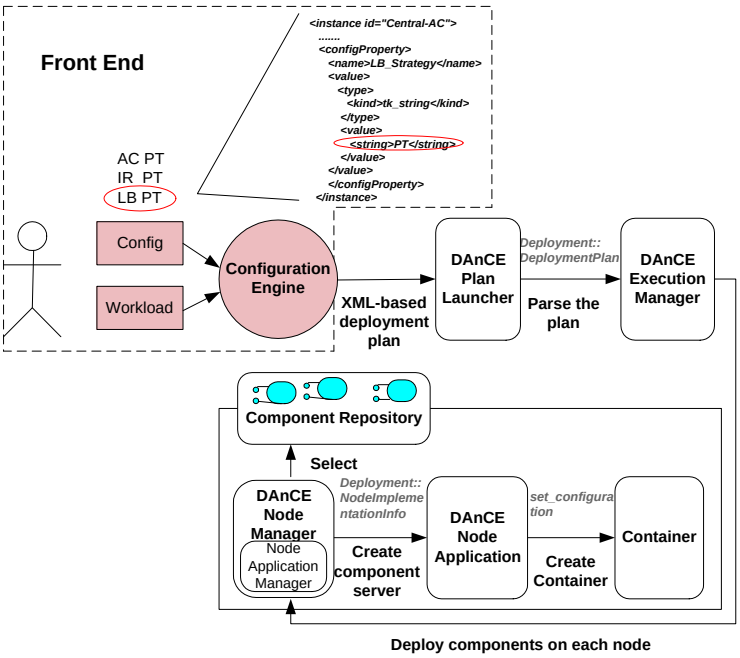

Fig. 3. Dynamic Configuration Process

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 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 3.2. Since a developer might specify incompatible service configuration combinations, our approach should be able to detect and disallow them. Finally, if no configuration file is provided or it omits configuration information, our configuration engine 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 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 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 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 KURT-Linux operating system. One of the Pentium-IV 2.5GHz machines is used as a central task manager where the AC and LB components are deployed. The other five machines are used as the application processors on which TB, F/I Subtask, Last Subtask and IR 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 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. 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 by the total utilization of all jobs requesting admission. 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 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 Figure 4 show the average (mean) results over the 10 task sets. As is shown in Figure 4, 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_*)

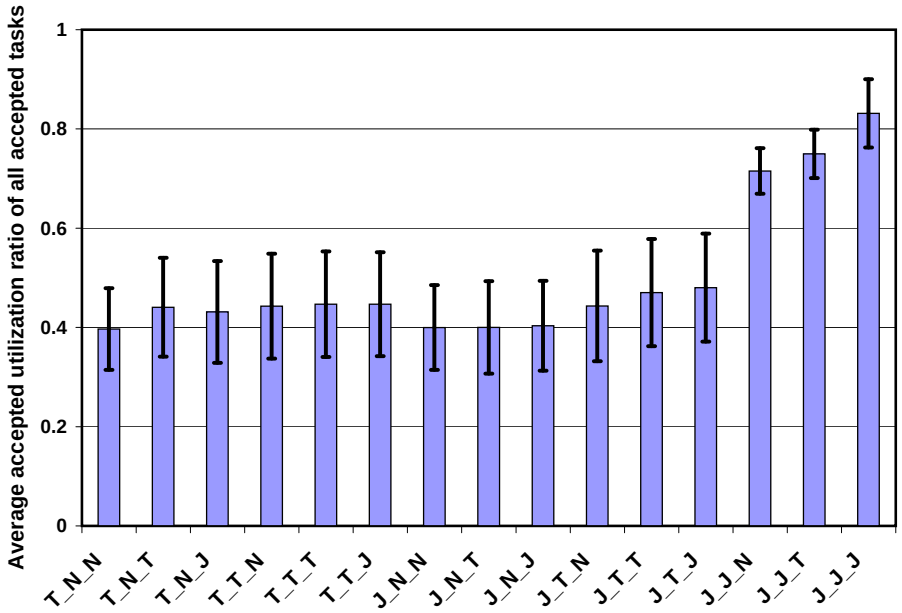

Fig. 4. Accepted Utilization Ratio

or not at all (*_N_*). This is because IR per job removes the contribution of all completed periodic jobs 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 significantly) 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 show the potential benefit of the LB component, we designed another experiment that is described 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 processors may experience heavy load. For example, in an industrial control system, a blockage in an automated assembly line 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 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 replica sitting on one processor in the second group.

Each of 10 task sets was run for the 15 different valid combinations, and for each combination we then averaged the utilization acceptance ratio over the 10 results. These 15 combinations can be divided into 5 sets. Each sets contains three combinations represented by three adjacent bars in Figure 5. In each set, we kept the admission control and the idle resetting strategies the same, but changed the load balancing strategy from none to per task, then to per job. 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 vs. per job.

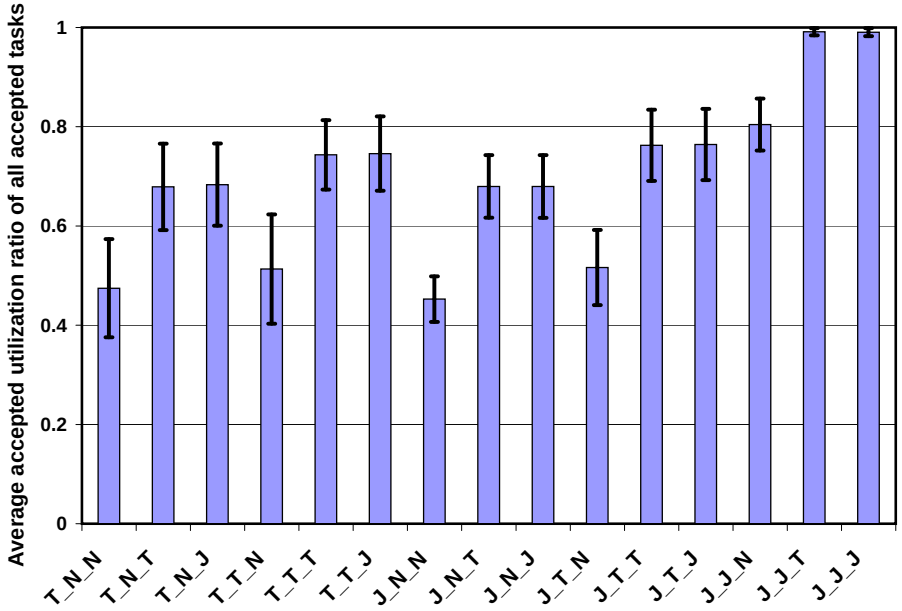

Fig. 5. Load Balancer Strategy Comparison

From these two experiments, we found that configuring different strategies according to application characteristics can have a significant impact on the performance of a real-time system with aperiodic and periodic tasks. 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.

5.3 Overheads of Service Components

To evaluate the efficiency of our component-based middleware services, we measured overheads using 3 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 5.1, 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 component TE1, after which AC and LB components run the task in component TE1 or re-allocate it to another TE component, TE2. Figure 6 shows how the total delay for each service includes the costs of operations located in several components. Figure 7 lists the operation numbers shown in Figure 6 to provide a detailed accounting of the delays resulting from different combinations of service configurations.

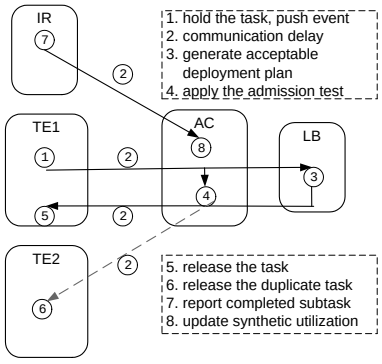

Fig. 6. Sources of Overhead/Delay

	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

Fig. 7. Overheads of Service Configurations (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 LB component 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 task manager processor (steps 3 and 4) 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 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 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 processor must be counted in the overall delay. However, 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 Figure 7. From the results in Figure 7, we can see that all of the delays induced by our configurable components are less than 2 ms, which is acceptable for many distributed real-time systems. However 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 3.2.

6 Related Work

Component Middleware: The architectural patterns used in the CORBA Component Model (CCM) [23] are also used in other popular component middleware technologies, such as J2EE [24, 25]. Among the existing component middleware technologies, CCM is the most suitable for distributed real-time applications since CORBA is the only standards-based COTS middleware that explicitly consider the QoS requirements of distributed real-time systems. In addition to CIAO, a number of CCM implementations are available, including OpenCCM [26], K2 Container [27], MicoCCM [28] and Qedo [29]. Clarke et al. [30] described the OpenCOM component model which can be used to construct a full middleware platform. Jordan et al. [31] provided a flexible and extensible framework for managing resources across a broad spectrum, from low-level resources like CPU time to higher-level resources such as database connections.

QoS-aware middleware: Quality Objects (QuO) [32, 33] 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 in distributed real-time applications. A Qosket is a unit of encapsulation and reuse for QuO systemic behaviors. In comparison to CIAO, Qoskets and QuO emphasize dynamic QoS provisioning where CIAO emphasizes static QoS provisioning and integration of various mechanisms and behaviors during different stages of the development lifecycle. The dynamicTAO [34] project applies reflective techniques to reconfigure Object Request Broker (ORB) components at run-time. Similar to dynamicTAO, the Open ORB [35] project also aims at highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements. Zhang et al. [36] use aspect-oriented techniques to improve the customizability of the middleware core infrastructure at the ORB level. Schantz et al. [37] show how priority- and reservation-based OS/network QoS management mechanisms can be coupled with standards-based, off-the-shelf middleware to better support dynamic distributed real-time applications with stringent end-to-end real-time requirements.

QoS-aware component Middleware: Component middleware's container architecture enables meta-programming of QoS attributes in component middleware. For example, aspect-oriented techniques can be used to plug in different systemic behaviors [38]. This approach is similar to CIAO in that it provides mechanisms to inject aspects into systems at the middleware level. de Miguel's work [39] further develops the state of the art in QoS-enabled containers by extending a QoS EJB container interface to support a QoSContext interface that allows the exchange of QoS-related information among component instances. To take advantage of the QoS-container, a component must implement QoSBean and QoSNegotiation interfaces. However, this requirement increases dependence among component implementations. The QoS Enabled Distributed Objects (Qedo) [40] project is another effort to make QoS support an integral part of 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 where QoS is part of the functional requirements, it tightly couples the QoS provisioning and adaptation behaviors into the component implementation, which may limit the reusability of the component. In comparison, CIAO explicitly avoids this coupling and composes the QoS aspects into applications declaratively. The OpenCCM [26] project is a Java-based CCM implementation. The OpenCCM Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. OpenCCM and its DCI infrastructure, however, do not support key QoS aspects for distributed real-time systems, including real-time QoS policy configuration and resource management. There have been several other efforts to introduce of QoS in conventional component middleware platforms. The FIRST Scheduling Framework (FSF) [41] proposes to compose several applications and to schedule the available resources flexibly while guaranteeing hard real-time requirements. A real-time component type model [42], which integrates QoS facilities into component containers also was introduced, based on the EJB and RMI specifications. A schedulability analysis algorithm [43] 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.

7 Conclusions

The work presented in this paper represents a promising step towards configurable admission control and load balancing support for a variety of distributed real-time applications with aperiodic and periodic tasks. We have designed and implemented configurable middleware components that provide effective on-line admission control and load balancing and can be easily configured and deployed on different processors. Our front-end 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 is well suited

for satisfying different applications with a variety of alternative characteristics and requirements, and (2) our component middleware services are appropriately efficient on a Linux platform.

References

1. Sha, L., et. al: Real Time Scheduling Theory: A Historical Perspective. The Journal of Real-Time Systems **10** (2004) 101–155
2. Zhang, Y., Lu, C., Gill, C., Lardieri, P., Thaker, G.: Middleware Support for Aperiodic Tasks in Distributed Real-Time Systems. In: RTAS. (2007)
3. Abdelzaher, T.F., Thaker, G., Lardieri, P.: A Feasible Region for Meeting Aperiodic End-to-end Deadlines in Resource Pipelines. In: ICDCS. (2004)
4. Institute for Software Integrated Systems: The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/ (Vanderbilt University)
5. Institute for Software Integrated Systems: Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/ (Vanderbilt University)
6. Deng, G., Schmidt, D.C., Gill, C., Wang, N., eds.: QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems. CRC Press to appear.
7. Object Management Group: Light Weight CORBA Component Model Revised Submission. OMG Document realtime/03-05-05 edn. (May 2003)
8. Object Management Group: Deployment and Configuration Specification. OMG Document ptc/2003-07-02 edn. (July 2003)
9. Strosnider, J., Lehoczky, J.P., Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. IEEE Transactions on Computers **44**(1) (1995) 73–91
10. Sha, L., Lehoczky, J.P., Rajkumar, R.: Solutions for some practical problems in prioritizing preemptive scheduling. In: RTSS. (1986)
11. Lehoczky, J.P., Sha, L., Strosnider, J.K.: Enhanced aperiodic responsiveness in a hard real-time environment. In: RTSS. (1987)
12. Sprunt, B., Sha, L., Lehoczky, L.: Aperiodic task scheduling for hard real-time systems. The Journal of Real-Time Systems **1**(1) (1989) 27–60
13. Ramos-Thuel, S., Lehoczky, J.P.: On-line scheduling of hard deadline aperiodic tasks in fixed-prioriry systems. In: RTSS. (1993)
14. Ramos-Thuel, S., Lehoczky, J.P.: Algorithms for scheduling hard aperiodic tasks in fixed priority systems using slack stealing. In: RTSS. (1994)
15. Lehoczky, J.P., Thuel, S.R.: An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In: RTSS. (1992)
16. Davis, R.I., Tindell, K., Burns, A.: Scheduling slack time in fixed priority preemptive systems. In: RTSS. (1993)
17. Spuri, M., Buttazzo, G.: Scheduling Aperiodic Tasks in Dynamic Priority Systems, Real-Time Systems. The Journal of Real-Time Systems **10**(2) (1996)
18. Andersson, B., Ekelin, C.: Exact Admission-Control for Integrated Aperiodic and Periodic Tasks. In: RTAS. (2005)
19. Harrison, T.H., Levine, D.L., Schmidt, D.C.: The design and performance of a real-time CORBA event service. In: Proceedings of OOPSLA. (1997)
20. Object Management Group: CORBA Components. OMG Document formal/2002-06-65 edn. (June 2002)
21. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus (October 2004)

22. Gokhale, A.: Component Synthesis using Model Integrated Computing. www.dre.vanderbilt.edu/cosmic (2003)
23. Volter, M., Schmid, A., E.Wolff: Server Component Patterns – Component Infrastructures illustrated with EJB. Wiley & Sons, New York (2002)
24. Volter, M., Schmid, A., Wolff, E.: Server Component Patterns: Component Infrastructures Illustrated with EJB. Wiley Series in Software Design Patterns, West Sussex, England (2002)
25. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall (2001)
26. Universite des Sciences et Technologies de Lille, F.: The OpenCCM Platform. corbaweb.lifl.fr/OpenCCM/ (2003)
27. iCMG: K2 Component Server. www.icmgworld.com (2003)
28. MICO: The MICO CORBA Component Project. www.fpx.de/MicoCCM/ (2000)
29. Qedo: QoS Enabled Distributed Objects. qedo.berlios.de (2002)
30. Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Springer-Verlag (2001) 160–178
31. Jordan, M., Czajkowski, G., Kouklinski, K., Skinner, G.: Extending a J2EE. Server with Dynamic and Flexible Resource Management. In: Middleware. (2004)
32. Schantz, R., Loyall, J., Atighetchi, M., Pal, P.: Packaging Quality of Service Control Behaviors for Reuse. In: Proceedings of the 5^{th} IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Crystal City, VA, IEEE/IFIP (April/May 2002) 375–385
33. Zinky, J.A., Bakken, D.E., Schantz, R.: Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems **3**(1) (1997) 1–20
34. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The Case for Reflective Middleware. Communications of the ACM **45**(6) (June 2002) 33–38
35. Gordon S. Blair and Geoff Coulson and Anders Andersen and Lynne Blair and Michael Clarke and Fabio Costa and Hector Duran-Limon and Tom Fitzpatrick and Lee Johnston and Rui Moreira and Nikos Parlavantzas and and Katia Saikoski: The Design and Implementation of Open ORB 2. IEEE Distributed Systems Online **2**(6) (June 2001)
36. Zhang, C., Jacobsen, H.A.: Resolving Feature Convolution in Middleware Systems. In: OOPSLA. (2004)
37. R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali: Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In: Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms, Rio de Janeiro, Brazil, IFIP/ACM/USENIX (June 2003)
38. Conan, D., Putrycz, E., Farcet, N., DeMiguel, M.: Integration of Non-Functional Properties in Containers. Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP) (2001)
39. de Miguel, M.A.: QoS-Aware Component Frameworks. In: The 10^{th} International Workshop on Quality of Service (IWQoS 2002), Miami Beach, Florida (May 2002)
40. FOKUS: Qedo Project Homepage. http://qedo.berlios.de/
41. Aldea, M., Bernat, G., Broster, I., Burns, A., Dobrin, R., Drake, J.M., Fohler, G., Gai, P., Harbour, M.G., Guidi, G., Gutiérrez, J.J., Lennvall, T., Lipari, G., Martínez, J.M., Medina, J.L., Palencia, J.C., Trimarchi, M.: FSF: A Real-Time Scheduling Architecture Framework. In: RTAS. (2006)
42. de Miguel, M.A.: Integration of QoS Facilities into Component Container Architectures. In: ISORC. (2002)
43. Lorente, J.L., Lipari, G., Bini, E.: A Hierarchical Scheduling Model for Component-Based Real-Time Systems. In: WPDRTS. (2006)