Enhancing the Robustness of Distributed Real-Time Middleware via End-to-End Utilization Control

Xiaorui Wang Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
{*wang, lu*}*@cse.wustl.edu*

Xenofon Koutsoukos
Department of Electrical Engineering
and Computer Science
Vanderbilt University
Nashville, TN 37235
xenofon.koutsoukos@vanderbilt.edu

Abstract

A key challenge for distributed real-time and embedded (DRE) middleware is maintaining both system reliability and desired real-time performance in unpredictable environments where system workload and resources may fluctuate significantly. This paper presents FC-ORB, a real-time Object Request Broker (ORB) middleware that employs end-to-end utilization control to handle fluctuations in application workload and system resources. The contributions of this paper are three-fold. First, we present a novel utilization control service that enforces desired CPU utilization bounds on multiple processors by adapting the rates of end-to-end tasks within user-specified ranges. Second, we describe a set of middleware-level mechanisms designed to support end-to-end tasks and distributed multi-processor utilization control in a real-time ORB. Finally, we present extensive experimental results on a Linux testbed. Our results demonstrate that our middleware can maintain desired utilizations in face of uncertainties and variations in task execution times, resource contentions from external workloads, and permanent processor failure. FC-ORB demonstrates that the integration of utilization control, end-to-end scheduling and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

1 Introduction

Distributed real-time and embedded (DRE) applications have stringent requirements for end-to-end timeliness and reliability whose assurance is essential to their proper operation. In recent years, many DRE systems have become open to unpredictable operating environments where both system workload and platform may vary significantly at run time. For example, the execution of data-driven applications such as autonomous surveillance is heavily influenced by sensor readings. External events such as detection of an intruder can trigger sudden increase in system workloads. Furthermore, many mission-critical applications must continue to provide real-time services despite hardware failures, software faults, and cyber attacks.

While DRE middleware has shown promise in improving the real-time properties of many applications, existing middleware systems often do not work well in unpredictable environments due to their dependence on traditional real-time schedulability analysis. When accurate knowledge about workloads and platforms is not available, a DRE application configured based on schedulability analysis may suffer deadline misses or even system crash [18]. A critical challenge faced by application developers is to achieve *robust* guarantees on real-time performance in unpredictable environments. Since in DRE systems, an end-to-end application that violates its real-time properties is equivalent to (or sometimes even worse than) an application that does not perform its computation, utilization guarantees affect directly the availability of the end-to-end application.

This paper presents the design and empirical evaluation of an adaptive middleware called *FC-ORB* (Feedback Controlled ORB) that aims to enhance the robustness of DRE applications. The novelty of FC-ORB is the integration of end-to-end scheduling, adaptive QoS control, and fault-tolerance mechanisms that are optimized for unpredictable environments. Specifically, this paper makes three contributions.

- *End-to-End Real-Time ORB*: Our ORB service supports end-to-end real-time tasks based on the end-to-end scheduling framework [16]. The FC-ORB architecture is designed to facilitate efficient end-to-end adaptation and fault-tolerance in memory-constrained DRE systems.
- *End-to-End Utilization Control*: The utilization control service enforces desired CPU utilizations in a DRE system despite significant uncertainties in system workloads. The core of the utilization control service is a distributed feedback control loop that coordinates adaptations on multiple interdependent processors.
- *Adaptive Fault Tolerance*: FC-ORB handles processor failures with an adaptive strategy that combines reconfigurable utilization control and task migration. A

unique feature of our fault tolerance approach is that it can maintain *real-time* properties for DRE applications after a processor failure.

FC-ORB has been implemented and evaluated on a Linux platform. Our experimental results demonstrate that FC-ORB can significantly improve the end-to-end real-time performance of DRE middleware in face of a broad set of dynamics including uncertainties and fluctuations in task execution times, resource contention from external workloads, and processor failures. FC-ORB demonstrates that the integration of utilization control, end-to-end scheduling and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

The rest of the paper is organized as follows. Section 2 describes the design of the FC-ORB architecture. Section 3 presents the experimental results. Section 4 highlights the contributions of FC-ORB by comparing it with related works. Section 5 concludes the paper.

2 Design of the FC-ORB Architecture

In this section, we first introduce the end-to-end task model and scheduling framework supported by FC-ORB. We then describe the main components of FC-ORB: the end-to-end ORB service, the utilization control service, and the adaptive fault-tolerance mechanisms.

2.1 Applications

FC-ORB supports an end-to-end task model [16] employed by many DRE applications. An application is comprised of m periodic tasks $\{T_i | 1 \leq i \leq m\}$ executing on n processors $\{P_i | 1 \leq i \leq n\}$. Task T_i is composed of a chain of subtasks $\{T_{ij} | 1 \leq j \leq n_i\}$ which are implemented as a sequence of object operations on different processors. The invocation of a subtask $T_{ij} (1 < j \leq n_i)$ is triggered by its predecessor T_{ij-1} through a remote operation request. A non-greedy synchronization protocol called release guard [31] is used to ensure that the interval between two consecutive releases of the same subtask is not less than its period. Hence, all the subtasks of a periodic task share the same rate as the first subtask. In FC-ORB, the rate of a task (and all its subtasks) can be adjusted by changing the rate of its first subtask. An example DRE application with two end-to-end tasks running on three processors is shown in Figure 1.

Our application model has two important properties. First, while each subtask T_{ij} has an *estimated* execution time c_{ij} available at design time, its *actual* execution time may be different from its estimation and may vary at run-time. Such uncertainty is common for DRE systems operating in unpredictable environments. Second, the rate of a task T_i may be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. This assumption is based on the fact that the task rates in many DRE applications (e.g., digital control [20][27], sensor update, and multimedia [4]) can be dynamically adjusted without causing system failure. A task running at a higher rate contributes a higher value to the application at the cost of higher utilization. For instance, although a digital control system usually has better control performance when it executes at a higher rate, it can usually remain stable when executing at a lower rate.

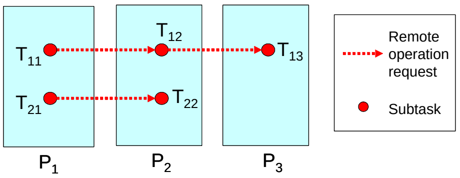


Figure 1. An example DRE application

Each task T_i is subject to an end-to-end soft deadline related to its period. FC-ORB implements the end-to-end scheduling approach [31] to meet task deadlines. The deadline of a task is divided into subdeadlines of its subtasks [9][22]. The release guard protocol is used to synchronize the execution of subtasks such that each subtask can be modeled as a periodic task. Hence, the problem of meeting the deadline is transformed to the problem of meeting the subdeadline of each subtask. A well known approach for meeting the subdeadlines on a processor is to ensure that its utilization remains below its schedulable utilization bound [13][15]. Therefore the end-to-end scheduling approach enables FC-ORB to meet end-to-end deadlines by controlling the utilizations of all processors in the system.

2.2 Middleware Support for End-to-End Tasks

In this subsection we first present how FC-ORB implements end-to-end tasks, and then introduce the priority management strategy.

2.2.1 Implementation of End-to-End Tasks

Figure 2 illustrates the FC-ORB implementation of the example DRE application shown in Figure 1. Each subtask is executed by a separate thread whose priority is decided by a priority manager. In Figure 2, each dashed box spanning from the application layer to the ORB core layer represents a subtask in Figure 1. Every subtask is associated with a separate Reactor [23] to create timeout events and to manage communication connections.

As shown in Figure 2, the first subtask of a task is implemented with a periodic ACE timer, a Reactor and a Connector [24]. The timer periodically triggers a local operation (e.g., a method of an object) which implements the functionality of this subtask. Following the execution of this operation, a one-way remote operation request is pushed

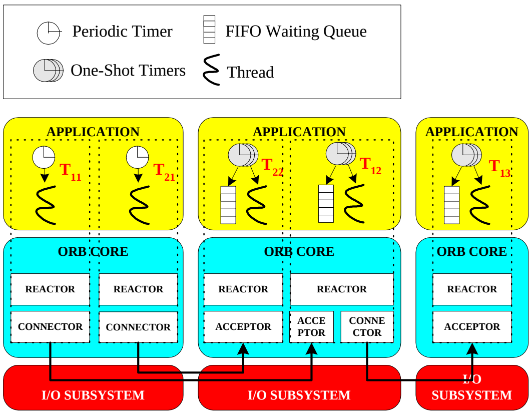


Figure 2. FC-ORB's end-to-end architecture

through the Connector to the succeeding subtask that is located on another processor. The succeeding subtask employs an Acceptor [24] to accept the request from its preceding subtask. Each pair of Connector and Acceptor maintains a separate TCP connection to avoid priority inversion in the communication subsystem. The release guard protocol enforces that the interval between two successive invocations of a same subtask is lower bounded by its period. Earlier research has shown that the release guard protocol can effectively reduce the end-to-end response time and jitter of tasks in DRE systems [31]. FC-ORB implements the release guard protocol with a FIFO waiting queue and one-shot ACE timers. Upon receiving a remote operation request, a subtask compares the current time with the last invocation time of this operation. Based on the release guard rules [31], the subtask either immediately invokes the requested operation or enqueues this request to the waiting queue if the request arrives too early. When the request is enqueued, a one-shot ACE timer is registered with the Reactor to trigger the requested operation at the time that equals the last invocation time plus the task's period. After the one-shot timer fires and the enqueued request is served, a remote operation request is sent to the next subtask in the end-to-end task chain. An end-to-end real-time task is finished when the execution of its last subtask is finished.

2.2.2 Priority Management

The integration of end-to-end scheduling and utilization control introduces new challenges to the design of scheduling mechanisms in ORB middleware. For instance, the rate adaptation mechanism adopted by FC-ORB and several other projects [18][19] may dynamically change the rates of end-to-end tasks. This may cause the middleware to change the priorities of all its subtasks, e.g., when the Rate Monotonic Scheduling (RMS) policy is used. To satisfy the special requirements posed by rate adaptation and end-to-end scheduling, our ORB service is configured with the *server-declared priority* model [25] and the *thread-per-subtask* concurrency architecture.

To support the server-declared priority model, FC-ORB implements a priority manager on each processor to assign priorities to local subtasks. The incoming requests from another processor are served by a thread with a real-time priority dictated by the priority manager located on the host processor. Currently the priority manager only supports the RMS policy, although the following discussions are also applicable to other rate- or deadline-dependent scheduling policies (note that task deadlines are usually related to their periods). There are several advantages of using server-declared priority model in the FC-ORB system. First, each processor is able to change thread priorities locally, based on the current rates of the subtasks located on it, so a processor only needs to know the local subtasks. This makes the system more scalable to large applications. Moreover, the server-declared model has less overhead because it does not have to adjust a thread's priority every time the priority of its predecessor subtask is changed, as it would do with the client-propagated model.

The thread-per-priority concurrency architecture has been adopted in existing DRE middleware (e.g., [26]). In this model,the same thread is responsible for executing all subtasks with a same priority. This is because the workload is assumed to use only a limited number of fixed task rates. However, this concurrency architecture is not suitable for rate adaptation. Due to rate adaptation, the rates and thus the priorities of subtasks vary dynamically at runtime. In such situations, the thread-per-priority architecture would require the ORB to dynamically move a subtask from one thread to another thread which can introduce significant overhead.

To avoid this problem FC-ORB implements the thread-per-subtask architecture that executes each subtask with a separate thread. FC-ORB adjusts the priorities of the threads only when the *order* of the task rates is changed. While the task rates may vary at every control period, the order of task rates often changes at a much lower frequency. Therefore, the thread-per-subtask architecture enables FC-ORB to adapt task rates in a more flexible way, with less overhead.

A potential advantage of the thread-per-priority architecture is that it may need fewer threads to execute applications. However, as FC-ORB is targeted at memory-constrained networked embedded systems that commonly have limited number of subtasks on a processor, each subtask can be easily mapped to a thread with a unique native thread priority even in a thread-per-subtask architecture.

2.3 End-to-End Utilization Control Service

FC-ORB allows users to specify a set of application parameters in a configuration file that is used to initialize the middleware when the system is started. Configuration parameters include the desired CPU utilization on each processor, and the allowed range of rate for each real-time task. The utilization control service dynamically enforces the desired CPU utilizations on all processors by adapting the rates of real-time tasks within the specified ranges, despite significant uncertainties and fluctuation in system workload and platform. Therefore, to guarantee end-to-end deadlines, the application users only need to specify the utilization reference of each processor to a value below its schedulable utilization bound.

In the rest of this subsection we first give an overview of the feedback control loop of the utilization control service, and then describe each component of the loop in detail.

2.3.1 Feedback Control Loop

The utilization control service implements the EUCON algorithm [19] as a distributed feedback control loop in the middleware. As shown in Figure 3, the feedback control loop is composed of a utilization monitor, a rate modulator and a priority manager on each processor, and a centralized controller.

As shown in Figure 3, the three components of the feedback control loop on an application processor (i.e., a processor executing applications and the ORB) are executed by a separate thread called the *control thread*. This control thread has the highest priority in the middleware system so that the feedback control loop can be executed in overload conditions, when it is needed most. The controller is implemented as an independent process that can be deployed on a separate processor or on an application processor. The controller also serves as a coordinator of the FC-ORB system. Every application processor in the system tries to connect with the controller through a TCP connection (called *feedback lane*) when the node is started. Once all application processors are connected to the controller, the whole system starts to run the configured application.

The feedback control loop is invoked in the end of every sampling period. It works as follows: (1) the utilization monitor on each processor sends its utilization in the last sampling period to the controller; (2) the controller collects the utilizations from all processors, computes the new task rates, and sends the new task rates to the rate modulators on all processors where the tasks are running; (3) the rate modulators on processors that host the first subtasks of tasks change the rates of the first subtasks according to the input from the controller; and (4) the priority manager on each processor check and adjust the thread priorities based on the new task rates if necessary.

2.3.2 Control Components

We now present the details of each utilization control component.

- **Controller:** The controller is implemented as a single-thread process. It employs a Reactor to interact with all processors in the system. Each time its periodic timer fires, it sends utilization requests to all application processors through the feedback lanes. The incoming replies are registered with the Reactor as events to be handled asynchronously. This enables the controller to avoid being blocked by an overloaded application processor. After it collects the replies from all processors, it executes a *Model Predictive Control* (MPC) algorithm proposed in [19] to calculate the new task rates. Then, for each task whose rate needs to be changed, the controller sends the task's new rate to all processors that host one or more subtasks of the tasks whose rates have been changed. If a processor does not reply in an entire control period, its utilization is treated as 100%, as the controller assumes this processor is saturated by its workload.

- **Utilization Monitor:** The utilization monitor uses the /proc/stat file in Linux to estimate the CPU utilization in each sampling period. The /proc/stat file records the number of jiffies (usually 10ms in Linux) when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle task, since the system starts. At the end of each sampling period, the utilization monitor reads the counters, and estimates the CPU utilization as 1 minus the number of jiffies used by the idle task in the last sampling period divided by the total number of jiffies in the same period.

- **Rate Modulator:** A Rate Modulator is located on each processor. It receives the new rates for its remote invocation requests from the controller through the feedback lane, and resets the timer interval of the first subtask of each task whose invocation rate has been changed.

- **Priority Manager:** All processors in FC-ORB assign priorities to their subtasks based on a real-time scheduling algorithm (e.g., RMS). It is important to strictly enforce the scheduling algorithm to achieve desired real-time performance. However, as a result of rate adaptation, a task with a rate higher than another task could be assigned a lower rate in the next sampling period. Consequently, the priority of this task has to be adjusted at run-time. The priority manager on each processor checks the rate order of all subtasks on this processor. If the rate order of two or more subtasks is reversed, the priority manager reassigns the correct priorities for the threads of those subtasks.

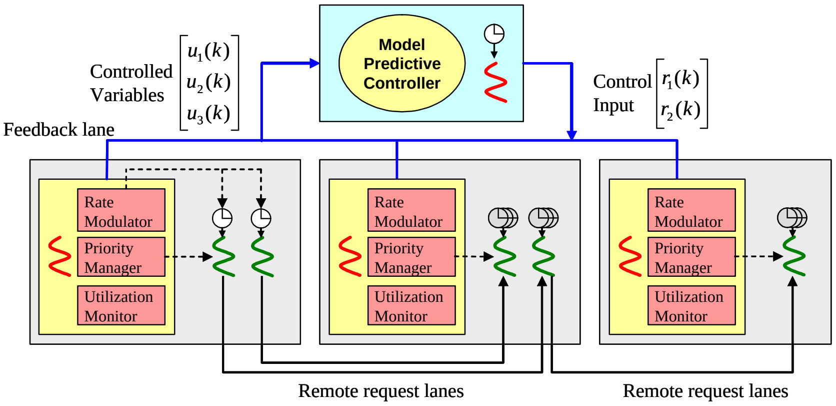


Figure 3. The distributed feedback control loop of the utilization control service

2.4 Fault Tolerance

A robust DRE middleware must maintain both reliability and real-time properties required by the applications despite partial system failures. Traditional fault-tolerance mechanisms usually focus on reliability aspects of the system based on *entity redundancy*. No single point of failure, transparent failover and transparent redirection and reinvocation are among the requirements of a fault-tolerant ORB [6]. However, less attention has been paid to maintaining desired real-time properties in face of faults.

Before describing the fault tolerance techniques in FC-ORB, we first introduce the fault model. FC-ORB is designed to handle persistent single processor failure. We assume that the communication between the remaining processors does not fail and the network is not overloaded. Our assumption regarding the network is reasonable for a common class of DRE systems where processors are connected with a switched/fast Ethernet LAN with sufficient bandwidth.

FC-ORB improves system robustness in terms of both reliability and real-time properties by integrating three complementary mechanisms. First, FC-ORB provides replication for subtasks and support transparent failover to backup subtasks located at different processors in face of processor failure. Second, after a processor fails, the remaining processors may experience dramatic workload increase due to the activation of the backup subtasks, which may cause them to miss deadlines or fail. A unique feature of FC-ORB is that it can effectively handle the workload increase via utilization control so that applications can maintain desired real-time properties despite processor failure. Finally, the FC-ORB controller can automatically reconfigure itself at runtime to rebuild its control model, in order to effectively control the DRE system whose deployment is changed due to processor failure.

In our replication mechanism, a subtask may have a backup subtask located on a different processor. For example, the subtask T_{13} shown in Figure 1 can have a backup subtask T'_{13} located on processor P_1. As a result, when processor P_3 fails because of hardware failure, the execution of subtask T_{13} is migrated to processor P_1 to continue automatically. Similar to the *COLD_PASSIVE* replication style used in Fault-Tolerant CORBA (FT-CORBA) [6], all subtasks are assumed to be stateless (except the connections between subsequent subtasks which are maintained by the middleware) so that the overhead of active state synchronization is avoided.

The failover mechanism works as follows. In the normal mode, each subtask pushes remote operation requests only to the primary instance of its successor. As a result, the backup instance does not receive any requests and its thread remains idle. After a processor fails, the predecessor of a subtask located on the failed processor detects the communication failure based on the underlying socket read/write errors. The predecessor immediately switches the connection to the backup instance of its successor and sends the remote operation requests to it. In the case when the failed processor hosts the first subtask of a task, the controller activates the backup instance of the subtask. Consequently, the execution of the end-to-end tasks is resumed after a transient interruption.

As a part of the fault-tolerant support, the controller in the utilization control service has been designed to be self-configurable. This is important because the control algorithm relies on knowledge about the subtask allocation in order to compute correct task rates [19]. When the controller detects communication failure with a processor in the system, it first cancels the periodic timer to pause the feedback control loop. In its internal control model, it then removes the failed processor and moves the subtasks located on the failed processor to the corresponding backup processors. After rebuilding the control model, the controller re-initializes itself and restarts the timer to resume the feedback control loop.

A disadvantage of the centralized control scheme is that the controller becomes a single point of failure. To mitigate this problem, FC-ORB can be easily extended to replicate the controller as well. In the extension, FC-ORB can actively maintain the state consistency between the primary controller and the backup controller, in a way similar to the *ACTIVE* replication style used in FT-CORBA [6]. When the controller executes in replicated mode, all processors send their CPU utilizations to both the primary and the backup controllers at every sampling instant. The backup controller performs control computation just like the primary controller. The difference is that the backup controller does *not* send the resultant new task rates to any processor. Instead, it uses this method to keep the state variables in the backup controller consistent with the primary controller. The primary and backup controllers can exchange heartbeat messages in every sampling period. Once the backup controller stops receiving heartbeats from the primary controller, the backup controller takes over the utilization control service. This feature will allow FC-ORB to maintain control of the entire system even after controller failures.

2.5 Implementation

FC-ORB 1.0 has been implemented in C++ using ACE 5.4 on Linux. FC-ORB is based on the FCS/nORB middleware [18] which integrates a *single-processor* feedback control scheduling service and a light-weight real-time ORB middleware called nORB [30]. FC-ORB is specialized for memory-constrained DRE systems by supporting a smaller set of features than general-purpose DRE middleware such as TAO. The entire FC-ORB middleware (excluding the code in ACE library and IDL library) is implemented in 7017 lines of C++ code. The controller is implemented in 1995 lines of C++ code. FC-ORB currently implements the control algorithm based on the constrained least square solver (*lsqlin*) in MATLAB. The controller process opens a MATLAB process at start time. In the end of each sampling period, the controller collects the utilizations from application processors and calls the solver in MATLAB with the utilizations as parameters. The solver computes the control input and return it to the controller. We choose the MATLAB solver as a proof of concept because it is a highly optimized and widely used solver. We plan to replace MATLAB with a native implementation of the solver in the future. All the code is open-source and can be downloaded from http://deuce.doc.wustl.edu/FCS_nORB/FC-ORB/.

3 Empirical Evaluation

In this section, we present the results of four sets of experiments run on a distributed testbed with five machines. Experiments I evaluates FC-ORB's performance when task execution times deviate from their estimations. Experiment II examines FC-ORB's capability of handling disturbances from external workloads. Experiment III tests FC-ORB's robustness in face of processor failure. Experiment IV compares the code size of FC-ORB with other embedded ORB middleware systems. Experimental results for varying execution times and run-time overhead can be found in an extended version of this paper [32].

3.1 Experimental Setup

All experiments are conducted on a testbed of five machines. All applications and the ORB service run on a Linux cluster composed of four Pentium-IV machines: Ron, Harry, Norbert and Hermione. Ron and Hermione are 2.80GHz and Harry and Norbert are 2.53GHz. All four machines are equipped with 512KB cache and 512MB RAM, and run KURT Linux 2.4.22. The controller is located on another Pentium-IV 2GHz machine with 512KB cache and 256MB RAM. The controller machine runs Windows XP Professional with MATLAB 6.0. The four machines in the cluster are connected via an internal switch and communicate with the controller machine through the departmental 100Mbps LAN.

All the experiments run a medium-sized workload that comprises 12 tasks (with a total of 25 subtasks). The tasks include 8 end-to-end tasks (tasks T_1 to T_8) and 4 local tasks. Figure 4 shows how the 12 tasks are distributed on the 4 application processors. A processor failure incident on Norbert is emulated in Experiment III to test FC-ORB's fault-tolerance capability. Hence in Figure 4, we also show the configured backup subtasks for all subtasks on Norbert that belong to an end-to-end task. There is no backup subtask for local task $T_{11,1}$ as we assume that the local task is specific to Norbert.

The subtasks on each processor are scheduled by the RMS algorithm [15]. Each task's end-to-end deadline is $d_i = n_i/r_i(k)$, where n_i is the number of subtasks in task T_i and $r_i(k)$ is the current rate of T_i. Each end-to-end deadline is evenly divided into subdeadlines for its subtasks. The resultant subdeadline of each subtask T_{ij} equals its period, $1/r_i(k)$. Hence the schedulable utilization bound of RMS [15], $B = m(2^{1/m}-1)$ is used as the utilization set point on a processor, where m is the number of subtasks (including backup subtasks) on this processor. Specifically, the utilization set points for the four experiment processors are: Ron (72.4%), Harry (72.4%), Norbert (74.3%) and Hermione (72.4%). All (sub)tasks meet their (sub)deadlines if the desired utilization on every processor is enforced. The sampling period of the utilization control service is $T_s = 4$ seconds.

To evaluate the robustness of FC-ORB when execution times deviate from the estimations, the execution time of each subtask T_{ij} can be changed by tuning a parameter called the *execution-time factor*, $etf_{ij}(k) = a_{ij}(k)/c_{ij}$, where a_{ij} is the actual execution time of T_{ij}. The execution time factor (*etf*) represents how much the actual execution time of a subtask deviates from the estimation. The *etf*

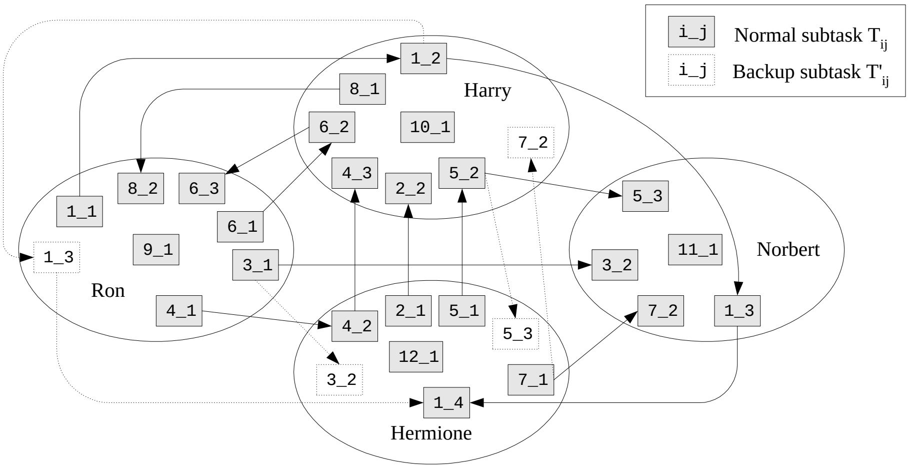


Figure 4. A medium size workload

(and hence the actual execution times) may be kept constant or changed dynamically in a run. In the following we use *inversed etf* (*ietf*) because DRE systems commonly have undesired oscillation when execution times are underestimated (i.e. $etf > 1$). Specifically, $ieft_{ij}(k) = 1/etf_{ij}(k)$.

We compare FC-ORB against a baseline called OPEN. In OPEN, the utilization control service of FC-ORB is turned off and the middleware becomes a representative real-time ORB without control. OPEN uses a typical open-loop approach to assign task rates based on *estimated* execution time to achieve the desired utilizations. OPEN results in desired utilization when estimated execution times are accurate (i.e., when $ietf = 1$). However, it causes underutilization when execution times are overestimated (i.e., $ietf > 1$), and over-utilization when execution times are underestimated (i.e., $ietf < 1$). This is a common problem faced by application developers because it is often difficult to estimate a tight bound on execution times, especially in unpredictable environment where execution times are heavily influenced by the value of sensor data or user input.

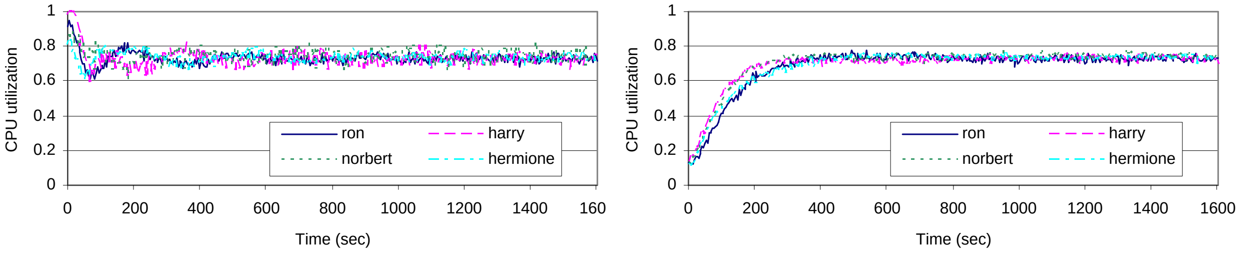
3.2 Experiment I: Uncertain Execution Times

In this subsection we evaluate FC-ORB's performance when task execution times deviate from the estimations. In each run of this experiment, all subtasks share a fixed execution-time factor (*ietf*).

First, we run experiments for OPEN which chooses task rates based on estimated execution times so that the estimated utilizations of all processors equal their set points. While the system achieves the desired utilizations in the ideal case when $ietf = 1$, all processors freezes when we set the *ietf* to 0.5. This is not surprising, because the actual execution time of every subtask in the system is *twice* its estimated execution time when $ietf = 0.5$. Consequently, the requested utilization on each processor is about 145% (twice of the desired utilization). Since all FC-ORB threads run at real-time priorities that are higher than the kernel priority on Linux, no kernel activities are able to execute causing the system to crash. This result shows that uncertainties in workloads can significantly degrade the robustness of applications on DRE middleware. On the other hand, the utilizations of all processors drop to only around 18% under OPEN when the actual execution times are only a *quarter* of their estimations ($ietf = 4$). This results in a extremely underutilized system and unnecessarily low task rates.

In contrast, FC-ORB achieves the desired utilizations on all processors even when execution times deviate significantly from the estimations. Figure 5(a) shows the utilizations for FC-ORB when the average execution time of every subtask is twice its estimation. In the beginning, all processors are overutilized because of the initial task rates. The utilization control service quickly decreases the task rates until the utilizations of all processors converge to the desired levels in around 400 seconds. Figure 5(b) shows the utilizations of all processors when the execution time of every subtask is severely overestimated ($ietf = 4$). In this case, all processors are initialized underutilized due to the low execution times. FC-ORB then increases the task rates until the utilizations of all processors converge to the set points roughly at 500 seconds. In this experiment, the utilization control service successfully prevents the system from crashing and underutilization via rate adaptation.

To examine FC-ORB 's performance under different execution time factors, we plot the mean and standard deviation of utilization on Harry during each run in Figure 6. Every data point is based on the measured utilization $u(k)$ from time 1200 seconds to 1600 seconds to exclude the transient response at the beginning of each run. FC-ORB consistently achieves the desired utilizations for all tested execution-time factors within the $ietf$ range $[0.5, 10]$ which corresponds to 20 times increase in execution times. This result shows that FC-ORB can provide robust guarantees on system reliability and real-time performance under a wide range of operating conditions. Interestingly, when the *ietf* is lower or equal to 0.33, the system freezes due



(a) ietf = 0.5

(b) ietf = 4

Figure 5. CPU utilizations under FC-ORB when task execution times deviate from estimations

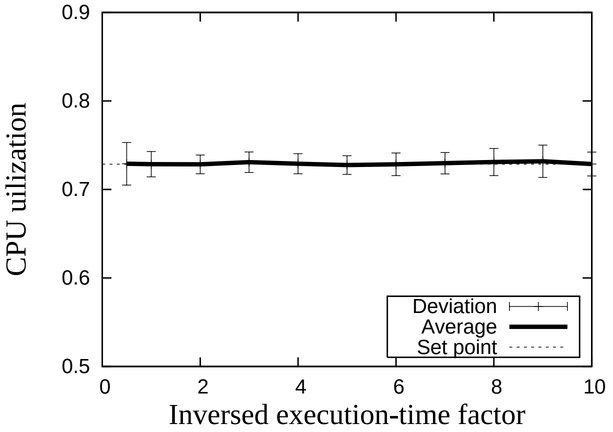


Figure 6. The average and deviation of CPU utilization on Harry under different *ietf*

to the extremely high utilization in the beginning of the run. Even though the control thread runs at highest real-time priority, the communication subsystem of Linux runs only at kernel priority. Therefore, the control thread of FC-ORB is blocked on communication because the Linux kernel is preempted by the middleware threads. As a result, the system fails to recover promptly from overload when the *ietf* is equal to or lower than 0.33, even with the help of FC-ORB. In addition, as observed in [19], the EUCON algorithm can cause performance oscillation when execution times are underestimated ($ietf < 1$). Therefore, application developers should use pessimistic estimations of task execution times in FC-ORB . A fundamental advantage of FC-ORB is that it does not cause system underutilization even when task execution times are severely overestimated.

3.3 Experiment II: External Disturbances

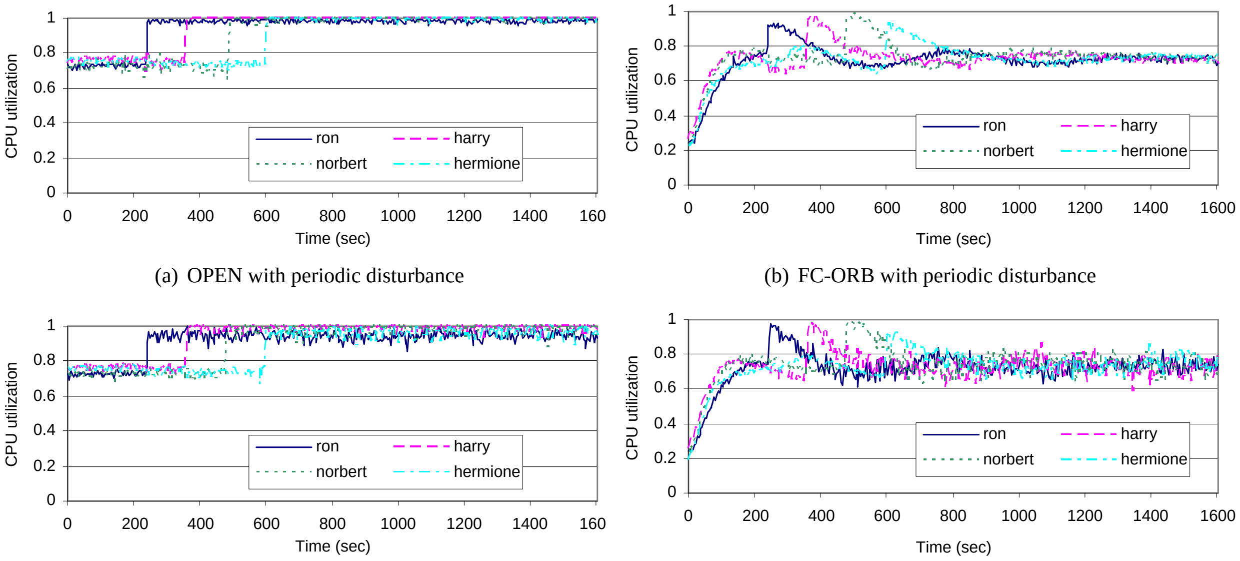
We now evaluate FC-ORB under resource contention from external workloads that are not controlled by FC-ORB. Such external disturbances may be caused by a variety of sources including (i) processing of critical events that must be executed at the cost of other tasks, (ii) varying workload from a different subsystem (e.g., legacy software from a different vendor), and (iii) software faults or adversarial cyber attacks. To stress-test FC-ORB, we emulate the external disturbances using a high priority real-time process to compete with FC-ORB for CPU resource. To investigate the robustness of FC-ORB we create both periodic and aperiodic disturbances. In the first set of runs, the external process *periodically* invokes a function with a *fixed* execution time of 100ms every 500ms. In the second set of runs, the external process *aperiodically* invokes another function with a *random* execution time. Both the request interarrival time and the execution time follow exponential distributions with mean values of 50ms and 10ms, respectively.

The workload controlled by FC-ORB has an $ietf = 2$. Here we manually configure the task rates in OPEN such that the workloads achieve the desired utilizations without the external disturbances. As shown in Figure 7(a), the system does achieve the required performance initially. However, at time 240sec, 360sec, 480sec and 600sec, the external task is activated sequentially on Ron, Harry, Norbert and Hermione. Consequently, the utilizations of all processors are raised to 100%. In contrast to OPEN, Figure 7(b) shows that FC-ORB successfully maintains the desired utilizations and thus tolerates the external resource contention. Similar situations occur for aperiodic disturbance, except that in this case, both OPEN and FC-ORB have higher fluctuation. Despite noise introduced by the aperiodic requests, FC-ORB still successfully maintains the CPU utilization under 80% most of the time and achieves the desired CPU utilizations on average.

3.4 Experiment III: Processor Failure

In this experiment we evaluate FC-ORB's ability to recover from processor failure. At 800 seconds, we emulate the failure of Norbert by using the Linux *kill* command to eliminate the process which carries FC-ORB and the application. The CPU utilization of Norbert immediately drops to almost zero because no other application is running on Norbert. All subtasks on Norbert have backup subtasks located on other processors as shown in Figure 4, except the local task $T_{11,1}$. Their preceding subtasks on other processors detect the communication failure with Norbert and then redirect the remote operation requests to the backup subtasks. Hence, the load of Norbert is distributed to the other 3 processors in the system.

As demonstrated in Figure 8, the CPU utilizations of the



(a) OPEN with periodic disturbance

(b) FC-ORB with periodic disturbance

(c) OPEN with aperiodic disturbance

(d) FC-ORB with aperiodic disturbance

Figure 7. CPU utilizations of all processors under external disturbances ($ietf = 2$)

other 3 processors increase simultaneously after the failure of Norbert. At the same time, the controller on the control processor re-configures itself to rebuild its control model after it detects the communication failure with Norbert. Thanks to the utilization control service, the high utilizations on the other 3 processors quickly converge to the desired utilization bounds within 100sec which ensures desired end-to-end real-time performance. Our results demonstrate that the system successfully recovers from a processor failure of a processor and the utilization of the remaining processors converges to a desirable state that ensures the real-time properties of the end-to-end application.

The fault injection using the *kill* command allows us to focus on the robustness of the utilization control service rather than the error detection method. Error detection is a complementary problem to the FC-ORB adaptation for error recovery. Our experimental evaluation of the FC-ORB robustness can be extended to more realistic processor crash failures assuming an appropriate error detection method. The time required for error recovery will include both the time needed for error detection and the convergence of the utilization control service. Formally evaluating the availability of the distributed application requires the definition of an appropriate benchmark [1][21], and is a subject of future work.

3.5 Experiment IV: Code Size

As FC-ORB is targeted at embedded systems, code size becomes a very important part of the overall memory footprint since typically all code of an embedded system is preloaded into its memory before execution. We compare the code size of FC-ORB with two other real-time embedded middleware called nORB [30] and FCS/nORB [18]. nORB is a light-weight real-time ORB based on a client/server architecture. It does not support end-to-end tasks. FCS/nORB integrates a feedback control real-time scheduling service with nORB. Its key difference from FC-ORB is that its can only control the real-time performance (utilization or deadline miss ratio) of a *single* server. We choose nORB and FCS/nORB as baselines for comparison because they are also specialized for memory-constrained embedded systems. Earlier results [30] showed that nORB has significantly smaller code size than general-purpose DRE middleware such as TAO [26].

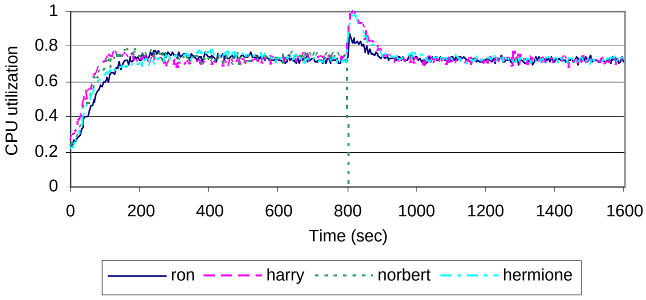


Figure 8. The CPU utilization of all processors while Norbert has a system failure ($ietf = 2$)

Figure 9 shows the code size of a same application implemented on different middleware. To have a fair comparison, we measure the average code size of the client and server for nORB and FCS/nORB. For FC-ORB , we measure the average code size on the four machines used in our experiments. Interestingly, Figure 9 shows that FC-ORB has the minimum code size, despite the fact that it provides more sophisticated services (e.g., release guard and

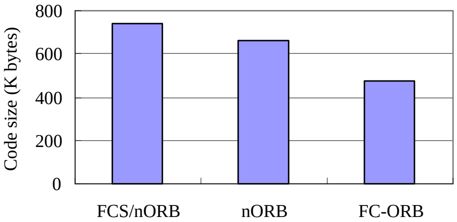


Figure 9. Code size comparison with other embedded middleware

end-to-end control) than nORB and FCS/nORB. The reduction in code size is attributed to simplification of the ORB implementation. For example, in FCS/nORB, each subtask is executed by a pair of threads connected through a FIFO queue. The separation of worker/timer threads and connection threads in FCS/nORB prevents the worker/timer threads from being blocked by communication subsystems when the network is overloaded. In FC-ORB, we choose to replace the thread pair with a single thread because FC-ORB is not designed to handle overloaded networks.

4 Related Work

Adaptive middleware is emerging as a core building block for DRE systems. For example, TAO [26], dynamicTAO [11], ZEN [10], and nORB [30] are adaptive middleware frameworks that can (re)configure various properties of ORB middleware at design- and run-time. Higher-level adaptive resource management frameworks, such as QuO [34], Kokyu [5] and RT-ARM [8], leverage lower-level mechanisms provided by ORB middleware to (re)configure scheduling, dispatching, and other QoS mechanisms in higher-level middleware. ORB services such as the TAO Real-Time Event Service [7] and TAO Scheduling Service [5] offer high-level services for managing reliability and real-time properties of interactions between application components. FC-ORB has several important features that distinguishes itself from earlier work on adaptive middleware. First, FC-ORB integrates the end-to-end scheduling service with a utilization control service. This integrated approach enables the middleware to meet end-to-end deadlines by dynamically controlling the utilizations on individual processors. Second, in contrast to earlier works that rely on heuristics-based adaptive techniques, FC-ORB implements control algorithms that has been rigorously designed and analyzed based on a control-theoretic approach. Finally, FC-ORB enhances traditional fault-tolerance mechanisms with utilization control techniques to handle processor failures.

Agilos [14] and ControlWare [33] were two earlier control-based middleware framework for QoS adaptation. However, they are targeted at multimedia and Internet servers instead of DRE applications.

Several other projects also applied control theoretic approaches to real-time systems. For example, Steere, et al., developed a feedback based CPU scheduler [29] that coordinated allocation of CPU cycles to consumer and supplier threads in a modified Linux kernel. Abeni, et al., presented analysis of a reservation-based feedback scheduler in [3]. Authors of [17] proposed a set of feedback control real-time scheduling algorithms that provide deadline miss ratio and utilization guarantees for single-processor systems. Feedback control real-time scheduling has also been extended to handle distributed systems [19][28]. For systems requiring discrete control adaptation strategies, hybrid control theory has been adopted to control state transitions among different system configurations [2][12]. A key difference between the work presented in this paper and the related work is that we describe the design and evaluation of a utilization control service in an ORB middleware, while the related work is based either on simulations or kernel implementations. ORB middleware is a particularly suitable layer for managing *end-to-end* adaptation in distributed systems since it operates at a broader (distributed) scope than stand-alone operating systems.

In our earlier work we studied EUCON [19] only through control-theoretic analysis and simulation results. FC-ORB implements and empirically evaluates the end-to-end utilization service on an ORB middleware and a physical testbed. Furthermore, we also extend the EUCON algorithm with controller reconfiguration and replication techniques for handling processor failures.

5 Conclusions

In summary, we have designed and implemented FC-ORB, a real-time ORB middleware with a novel end-to-end utilization control service. Our experiments on a physical testbed has shown that (1) FC-ORB can enforce desired utilizations on all processors in a DRE system, even when task execution times deviate significantly from their estimated values or vary significantly at run-time; (2) FC-ORB can survive considerable resource contention imposed by external disturbances; (3) FC-ORB enhances the robustness of real-time properties to processor failures; (4) the middleware layer instantiation of the end-to-end utilization control service only introduces a small amount of processing and memory overhead. These results demonstrate that the integration of end-to-end utilization control, fault-tolerance mechanisms, and end-to-end scheduling in ORB middleware is a promising approach to achieve robust real-time performance guarantees for DRE applications. In the future, we plan to enhance FC-ORB to incorporate other adaptation mechanisms such as admission control and task reallocation so that FC-ORB can be applied to a broader class of applications. An important research direction is to integrate FC-ORB with advanced error detection and fault tolerance techniques in order to handle more complex fault models.

Acknowledgements

This research was supported in part by NSF CAREER award (grant CNS-0448554) and DARPA Adaptive and Reflective Middleware Systems (ARMS) program (grant NBCHC030140). We would also like to thank the reviewers for their detailed feedback.

References

[1] DBench Project Final Report. `www.laas.fr/DBench/Final/DBench-Short-Final-report.pdf`, May 2004.

[2] S. Abdelwahed, S. Neema, J. P. Loyall, and R. Shapiro. A hybrid control design for QoS management. In *RTSS*, 2003.

[3] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *IEEE RTSS*, Dec. 2002.

[4] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *IEEE RTSS*, Dec. 1998.

[5] C. D. Gill, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems, special issue on Real-Time Middleware*, 20(2), Mar. 2001.

[6] A. S. Gokhale, B. Natarajan, D. C. Schmidt, and J. K. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing*, 7(4):331–346, 2004.

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

[8] J. Huang and R. Jha and W. Heimerdinger and M. Muhammad and S. Lauzac and B. Kannikeswaran and K. Schwan and W. Zhao and R. Bettati. RT-ARM: A real-time adaptive resource management system for distributed mission-critical applications. In *Workshop on Middleware for Distributed Real-Time Systems, RTSS*, 1997.

[9] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[10] R. Klefstad, D. C. Schmidt, and C. O'Ryan. Towards highly configurable real-time object request brokers. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 437–447, 2002.

[11] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, June 2002.

[12] X. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu. Hybrid supervisory utilization control of real-time systems. In *IEEE RTAS*, 2005.

[13] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadline. In *IEEE RTSS*, 1990.

[14] B. Li and K. Nahrstedt. A control-based middleware framework for QoS adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, Sept. 1999.

[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM, Vol. 20, No.1, pp. 46-61*, Jan. 1973.

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

[17] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1/2):85–126, July 2002.

[18] C. Lu, X. Wang, and C. Gill. Feedback control real-time scheduling in ORB middleware. In *IEEE RTAS*, May 2003.

[19] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Trans. Parallel and Distrib. Syst.*, 16(6):550–561, June 2005.

[20] P. Marti, G. Fohler, P. Fuertes, and K. Ramamritham. Improving quality-of-control using flexible timing constraints: metric and scheduling. In *IEEE RTSS*, 2002.

[21] J.-F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Trans. Computers*, 29(8):720–731, Aug. 1980.

[22] M. D. Natale and J. Stankovic. Dynamic end-to-end guarantees in distributed real-time systems. In *IEEE RTSS*, 1994.

[23] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: AddisonWesley.*, 1995.

[24] D. C. Schmidt. Acceptor and Connector: Design patterns for initializing communication services. *Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley.*, 1997.

[25] D. C. Schmidt and F. Kuhns. An overview of the real-time CORBA specification. *IEEE Computer*, 33(6):56–63, 2000.

[26] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A pattern-oriented object request broker for distributed real-time and embedded systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[27] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control system. In *IEEE RTSS*, Dec. 1996.

[28] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed systems. In *RTSS*, Dec. 2001.

[29] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158, 1999.

[30] V. Subramonian, G. Xing, C. D. Gill, C. Lu, and R. Cytron. Middleware specialization for memory-constrained networked embedded systems. In *IEEE RTAS*, 2004.

[31] J. Sun and J. W.-S. Liu. Synchronization protocols in distributed real-time systems. In *ICDCS*, 1996.

[32] X. Wang, C. Lu, and X. Koutsoukos. Enhancing the robustness of distributed real-time middleware via end-to-end utilization control. Technical Report WUCSE-2005-45, Washington University in St. Louis, 2005.

[33] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*, July 2002.

[34] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.