To appear in *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, ON, May 25–28, 2004.

Dynamic CPU Management for Real-Time, Middleware-Based Systems

Eric Eide Tim Stack John Regehr Jay Lepreau
University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

{eeide,stack,regehr,lepreau}@cs.utah.edu http://www.cs.utah.edu/flux/

***Abstract*— Many real-world distributed, real-time, embedded (DRE) systems, such as multi-agent military applications, are built using commercially available operating systems, middleware, and collections of pre-existing software. The complexity of these systems makes it difficult to ensure that they maintain high quality of service (QOS). At design time, the challenge is to introduce coordinated QOS controls into multiple software elements in a non-invasive manner. At run time, the system must adapt dynamically to maintain high QOS in the face of both expected events, such as application mode changes, and unexpected events, such as resource demands from other applications.**

In this paper we describe the design and implementation of a *CPU Broker* for these types of DRE systems. The CPU Broker mediates between multiple real-time tasks and the facilities of a real-time operating system: using feedback and other inputs, it adjusts allocations over time to ensure that high application-level QOS is maintained. The broker connects to its monitored tasks in a non-invasive manner, is based on and integrated with industry-standard middleware, and implements an open architecture for new CPU management policies. Moreover, these features allow the broker to be easily combined with other QOS mechanisms and policies, as part of an overall end-to-end QOS management system. We describe our experience in applying the CPU Broker to a simulated DRE military system. Our results show that the broker connects to the system transparently and allows it to function in the face of run-time CPU resource contention.

I. INTRODUCTION

To meet the requirements of the market, real-time and embedded software systems must increasingly be designed atop commercial, off-the-shelf (COTS) operating systems and middleware. These technologies promote rapid software development by allowing system developers to concentrate on their application logic rather than on low-level "infrastructural" code. In addition, commercial operating systems and middleware promote software quality by providing tested, efficient, and reliable implementations of low-level functionality. Finally, these technologies promote scalability across different types of embedded platforms, configurability of features and feature selection, and evolvability of the embedded software systems over time. These so-called "-ilities" are essential in a world where embedded system technologies change rapidly and where the high cost of software development must be amortized over several products, i.e., across the life cycle of a product family rather than the lifetime of a single product.

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreement F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

COTS operating systems and middleware are also increasingly required to support the development of distributed, real-time, embedded (DRE) systems. Many real-time systems are built containing multiple processors or processing agents, either tightly connected (e.g., within an automobile or aircraft) or loosely connected (e.g., multi-player networked games, sensor networks, and networked military systems). Middleware such as CORBA [1] promotes the development of these systems by providing high-level and scalable abstractions for communication between multiple processes. Real-time middleware, such as RT CORBA [2], also provides high-level and portable abstractions for scheduling resources for real-time tasks.

Even with modern middleware, however, it can be a significant software engineering challenge for system developers to design and build DRE systems that meet their real-time requirements. First, because the parts of an embedded software system must often be designed to be reusable across many products, the code that implements real-time behavior for any particular system must be decoupled from the "application logic" of the system's parts. Decoupling makes it possible to collect the real-time specifications for all of the system's parts in a single place — in other words, to modularize the real-time behavior of the system — but leads to the new problem of reintroducing that behavior into the software. Second, even if the implementation of real-time behavior is modularized, developers are challenged with specifying the desired behavior at all. It is a common problem for the execution times of parts of a system to be data-dependent, mode-dependent, configuration-dependent, unpredictable, or unknown. In a distributed real-time system, the sets of communicating tasks and available processor resources may not be known until run time, or may change as the system is running. In sum, the challenges of implementing real-time behavior in many systems include not only decoupling and modularizing of the behavior, but the ability to describe a variety of policies in a high-level and tractable manner, and ensuring that the system continues to operate (perhaps at reduced capacity) in the face of events that occur at run time, both expected and unexpected.

To address these challenges, we have designed and implemented a novel *CPU Broker* for managing processor resources within real-time systems. Our CPU Broker is a CORBA-based server that mediates between the multiple real-time tasks and the facilities of a real-time operating system, such as TimeSys Linux [3]. The broker addresses design-time challenges by connecting to its managed tasks in a non-invasive fashion and by providing an expressive and open architecture for specifying CPU scheduling policies. The broker can manage resources for both CORBA and non-CORBA applications. At run time, the broker uses feedback and other inputs to monitor resource usage, adjust allocations, and deal with contention according to a configured policy or set of policies. The broker is configured at run time through a command-line tool or via invocations on the CORBA objects within the broker: policies are easily set up and changed dynamically. Finally, the broker is designed to fit into larger, end-to-end architectures for quality of service (QOS) management. A single instance of the broker manages CPU resources on a single host, but because its architecture is open and extensible, the broker's policies can be directed by higher-level QOS systems like QuO [4]. This enables coordination of brokers on multiple hosts, coordination with other resource managers, and cooperation with application-level QOS such as dynamic adaptation strategies.

Our CPU Broker was designed to ensure that the CPU demands of "important" applications are satisfied insofar as possible, especially in the face of dynamic changes in resource requirements and availability, in the set of managed tasks, and in the relative importances of the tasks. We have evaluated the broker in these situations through micro-benchmarks and synthetic application scenarios. In addition, we have applied and evaluated the CPU Broker in the context of a simulated DRE military application. Our results show that the broker correctly allocates CPU resources in our synthetic tests and in a "real world" system of target-seeking unmanned aerial vehicles (UAVs). The broker connects to applications in a transparent fashion and improves the ability of the UAV system to identify targets in a timely manner.

The primary contributions of this paper are threefold. First, we describe our architecture for dynamic CPU management in real-time systems: an architecture that addresses the critical software engineering challenges of specifying and controlling real-time behavior in the presence of anticipated and unanticipated events. Second, we present our CPU Broker, which effectively implements our architecture atop a commercial RTOS and industry-standard middleware, enables non-invasive integration, and provides an open and extensible platform for CPU management. Finally, we demonstrate the use of our CPU Broker and evaluate its performance in both synthetic scenarios and a simulated DRE military application. Our results show that the broker approach can effectively address both the design-time and run-time challenges of managing real-time behavior in COTS-based real-time systems.

II. Related Work

A great deal of work has been done in the areas of feedback-driven scheduling, real-time middleware, and middleware-based QOS architectures. In this section we summarize representative work in each of these areas and compare it to the CPU Broker. In general, our focus has been to improve on previous work by addressing the practical and software engineering barriers to deploying adaptive, feedback-driven scheduling in modern COTS-based embedded and real-time systems. These goals are elaborated in our previous work [5].

In the area of feedback-driven scheduling, Abeni and Buttazzo [6] describe a *QoS Manager* that is similar to our CPU Broker. The QoS Manager handles requests from three types of real-time tasks: pseudo-proportional-share tasks, (periodic) multimedia tasks, and (aperiodic) event-driven tasks. Tasks are weighted with importance values, and the QoS Manager uses feedback from multimedia and event-driven tasks to adjust those tasks' scheduling parameters. Our work differs from theirs in three important ways. First, whereas the QoS Manager is implemented within the HARTIK research kernel, our CPU Broker is built atop a COTS operating system and industry-standard middleware. (In later work [7], Abeni et al. implemented and analyzed a feedback-driven scheduler for Linux/RK [8].) Second, our focus is on non-invasive approaches: Abeni and Buttazzo do not describe how to cleanly separate the feedback from the applications being controlled. Third, our CPU Broker is based on an open architecture, making it easy to implement new policies, inspect the broker, or otherwise extend it. The QoS Manager, on the other hand, has a more traditional, monolithic architecture. Similar differences distinguish our CPU Broker from Nakajima's adaptive QOS mapping system [9], which was used to control video streaming applications under Real-Time Mach.

In the area of real-time middleware, there has been significant work in both commercial standards and novel research. For instance, the Object Management Group has defined and continues to evolve standards for RT CORBA [2], [10]. These standards are concerned with issues such as preserving thread priorities between clients and servers in a distributed system and proper scheduling of I/O activities within an ORB. This is in contrast to our CPU Broker, which manages the resources of a single host, is feedback-driven, and which operates above the ORB rather than within it. RT CORBA is similar to our CPU Broker, however, in that both provide an essential level of abstraction above the real-time services of an underlying operating system: it would be interesting to see if the (priority-based) RT CORBA and the (reservation-based) CPU Broker abstractions could be used in combination in the future. Other researchers have incorporated feedback-driven scheduling into real-time middleware: for example, Lu et al. [11] integrated a *Feedback Control real-time Scheduling service* into nORB, an implementation of CORBA for networked embedded systems. Their service operated by adjusting the rate of remote method invocations on server objects: i.e., by adjusting the clients to match the resources of the server. Our CPU Broker, on the other hand, would adjust the resources available to the server in order to meet its clients' demands. Our approaches are complementary: a robust DRE system might use both client-side and server-side adaptation effectively, and our CPU Broker is open to integration with other QOS adaptation mechanisms. Finally, it should be noted that our CPU Broker can manage both middleware-based and non-middleware-based processes, in contrast to the systems described above.

While many middleware-based QOS architectures operate by mediating between applications and an operating system, other architectures are based on applications that can adapt themselves to changing resource availability. The *Dynamic* QOS *Resource Manager* (DQM) by Brandt et al. [12] is of this second type and is perhaps the most similar to our CPU Broker in both its goals and approach. Like our broker, DQM is implemented as a middleware server atop a commercial OS. It monitors a set of applications, and based on CPU consumption and availability, it tells those tasks to adjust their requirements. The primary difference with our CPU Broker is in the level of adaptation. Our broker changes (and enforces) tasks' CPU allocations by interfacing with an RTOS. DQM, on the other hand, requests (but cannot enforce) that applications switch to new "execution levels," i.e., operational modes with differing resource needs. DQM and the CPU Broker implement complementary approaches to CPU management, and it would be interesting to combine our broker with a framework for benefit-based, application-level adaptation like DQM. Several researchers, including Abeni and Buttazzo [13], have demonstrated the benefits of combining adaptive reservation-based scheduling with application-specific QOS strategies.

The CPU Broker uses QuO [4] to connect to CORBA-based applications in a non-invasive manner and to integrate with other QOS management services such as application-level adaptation. QuO provides an excellent basis for coordinating multiple QOS management strategies in middleware-based systems, both for different levels of adaptation and for different resource dimensions. For example, Karr et al. [14] describe how QuO can coordinate application-level and system-level adaptations, e.g., by dropping video frames and reserving network bandwidth. More recently, Schantz et al. [15] demonstrated that the distributed UAV simulation (described in Section V-C) could be made resilient to both communication and processor loads by applying network reservations in combination with our CPU Broker.

III. Design

The conceptual architecture of the CPU Broker is illustrated in Figure 1, which depicts a sample configuration of the broker. At the top of the figure are the set of tasks (e.g., processes) being managed by the broker. Under those, within the dashed box, are the objects that make up the CPU Broker. As described later in Section IV, these objects are normally located all within a single process, but they do not have to be. The broker mainly consists of two types of objects, called *advocates* and *policies*, that implement per-application adaptation and global adaptation, respectively.

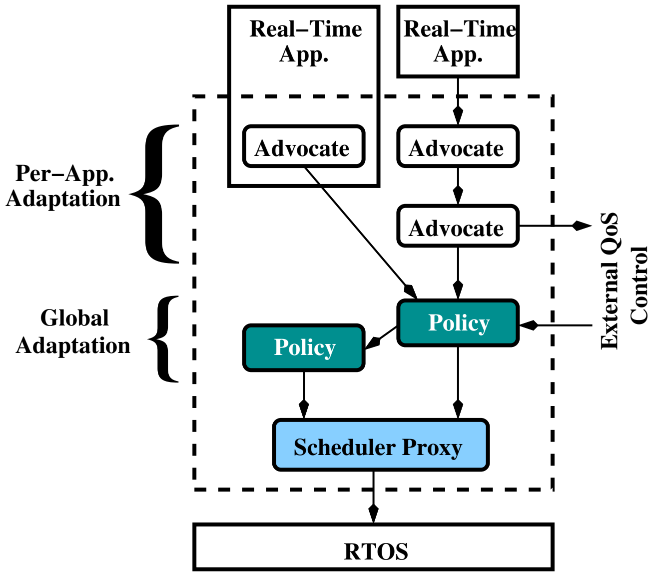


Fig. 1. Overview of the CPU Broker architecture

A. Advocates

As shown in Figure 1, every task under the control of the CPU Broker is associated with one or more *advocate* objects. The purpose of an advocate is to request CPU resources on behalf of a task. More generally, an advocate transforms an incoming request for resources into an outgoing request.

The primary input to an advocate is a request for a periodic CPU reservation: i.e., a period and an amount of CPU time to be reserved in each period. This is shown as the arrow entering the top of each advocate. On a regular basis — usually, at the end of every task cycle — the topmost advocate for a task receives a report of how much CPU time was consumed by the task since its last report: this is called the *status*. (The details of obtaining a task's status are described in Section IV.) The status amount may be more than what is currently reserved for the task: the CPU Broker normally manages "soft" reservations, which allow tasks to consume unreserved CPU resources in addition to their own reserves.

From this information, the topmost advocate decides how its task's reservation should be adjusted. The CPU Broker provides a family of different advocates, implementing different adaptation algorithms, and is an open framework for programmers who need to implement their own. A typical advocate works by producing a reservation request that closely matches its task's observed behavior. For instance, if the status amount is greater than the task's currently reserved compute time, the advocate would request a new reservation with an increased compute time that (better) meets the task's demand. If the status amount is smaller, on the other hand, then the requested compute time would be reduced. The details of the adaptation strategies are up to the individual advocates; for instance, different advocates may use historical data or may adapt reservations quickly or slowly. Advocates can also take input from a variety of sources in order to implement

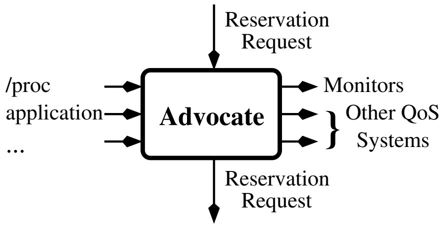


Fig. 2. Advocate

their policies, as illustrated in Figure 2. For instance, our architecture allows an embedded systems designer to deploy an advocate that observes application-specific data, such as mode changes, in order to make more predictive reservation requests. The different strategies that we have implemented are described in Section IV-C. In general, the strategies we have implemented for periodic tasks adjust a task's reservation so that the allocated compute time is within a small threshold above the task's demand, thereby allowing the task to meet its deadlines. Other implemented advocates are appropriate for controlling aperiodic and continuous-rate tasks, for example, by requesting user-specified fixed shares of the CPU.

The reservation that is requested by an advocate is called the *advice*. An advocate computes its advice and then invokes the next advocate in the chain, passing it both the task status and the computed advice.

Subsequent advocates are used to modify the advice or perform some side-effect: allowing advocates to be composed in this way greatly increases the CPU Broker's flexibility and allows existing advocates to be reused. For example, an advocate can be used to cap the resource request of a task, or alternately, ensure that the request does not fall below some minimum reserve. A side-effecting advocate might read or write information from other QOS management middleware, such as QuO. Another kind of side-effecting advocate is a "watchdog" that wakes up when some number of expected status reports have not been received from a monitored task. Reports are normally made at the end of task cycles, but this granularity may be too large, especially in the case of a task that has an unanticipated dynamic need for a greatly increased reservation. Without the intervention of a watchdog, the task might not be able to report its need until several task periods have passed — and several deadlines missed.[1] A watchdog can be used in these cases to mitigate the effects of dynamic workload increases: the watchdog wakes up, (possibly) inspects the state of its task, and requests an increased reservation on behalf of the task. In sum, composed advocates are a powerful mechanism for building complex behaviors from simple ones, for introducing and modularizing application-specific adaptations, and for conditioning the requests that are presented to the broker's contention policy object.

[1]In cases where missed deadlines are unacceptable, a system designer can easily configure the CPU Broker with different advocates to ensure that critical tasks have sufficient CPU reserves to process dynamic loads without missing deadlines. The essential point is that the CPU Broker enables system designers to choose the strategies that best meet the QOS needs of their applications.

B. Policies

The last advocate in the chain passes the task's status and the advice to a *policy* object that is responsible for managing the requests made on behalf of all tasks. A policy has two primary roles. First, it must resolve situations in which the incoming requests cannot all be satisfied at the same time, i.e., handle cases in which there is contention for CPU resources. Second, a policy must communicate its decisions to an underlying scheduler, which implements the actual CPU reservations.

The usual input to a policy object is a reservation request from one of the advocates in the CPU Broker. In response, the policy is responsible for re-evaluating the allocation of CPU resources on the host. Conceptually, a policy recomputes the reservations for all of the broker's managed tasks in response to an input from any advocate, but in practice, this recomputation is fast and most reports do not result in changes to the existing allocations. As with advocates, the CPU Broker provides a set of policy objects, each implementing different behaviors and conflict resolution strategies. In addition, if necessary, the implementer of a real-time system can implement and deploy a custom policy object within the broker's open framework. Most policies depend on additional data in order to be useful: for instance, the broker provides policies that resolve scheduling conflicts according to user-determined task importances. Dynamic changes to these importances — signaled via a (remote) method invocation on a policy object — will cause a policy to redetermine its tasks' allocations. The broker also provides a policy that partitions tasks into groups, where each group is assigned a maximum fraction of the CPU, and contention with each group is resolved by a secondary policy. The details of these policies, how they are set up, and how they can be changed dynamically are provided in Section IV.

Once a policy has determined the proper CPU reservations for its task set, it invokes a *scheduler proxy* object to implement those reservations. The scheduler proxy provides a facade to the actual scheduling services of an underlying RTOS. The RTOS (not the CPU Broker) is responsible for actually implementing the schedule and guaranteeing the tasks' reservations. If the RTOS rejects a reservation that was determined by the broker's policy, then the policy is responsible for modifying its request. (In practice, our implemented policies avoid making inadmissible reservation requests.)

The policy finishes with the scheduler and finally sends information about any changed reservations back to the advocates, which send the data up the advocate chains. An advocate may use this information to inform future requests, to cooperate with other QOS management frameworks, or to signal its application to adapt to its available CPU resources, for example. Eventually, a new CPU status report is received from one of the broker's managed tasks, and the process of the advocates and policies repeats.

IV. Implementation

In this section we describe how the design goals of the CPU Broker are met in its actual implementation. To achieve our goal of providing an open and extensible framework for dynamically managing real-time applications, we implemented the CPU Broker using CORBA. To achieve non-invasive integration with middleware-based real-time applications and other QOS management services, we used the QuO framework. Finally, to apply and demonstrate our architecture atop a commercial off-the-shelf RTOS, we implemented the broker for TimeSys Linux. The rest of this section describes how these technologies are used in our implementation, and also, the advocate and policy algorithms that we provide as part of the CPU Broker software.

A. Scheduling and Accounting

At the bottom of our implementation is TimeSys Linux [3], a commercial version of Linux with support for real-time applications. TimeSys Linux provides several features that are key to our implementation. First, the kernel implements reservation-based CPU scheduling through an abstraction called a *resource set*. A resource set may be associated with a periodic CPU reservation; zero or more threads are also associated with the resource set and draw (collectively) from its reservation. Second, the TimeSys kernel API allows a thread to manipulate resource sets and resource set associations that involve other threads, even threads in other processes. This makes it straightforward for the CPU Broker to manipulate the reservations used by its managed tasks. Third, whenever one thread spawns another, and whenever a process forks a child, the parent's association with a resource set is inherited by the child (by default). This makes it easy for the CPU Broker to manage the reservation of a task (process) as a whole, even if the task is internally multithreaded. Finally, TimeSys Linux provides high-resolution timers that measure the CPU consumption of threads and processes. These timers were essential in providing accurate reservations — and allowing high overall CPU utilization — in the real-time task loads we studied. To allow the CPU Broker to obtain high-resolution information about all of the threads in another process, we made a very small patch to the TimeSys Linux kernel to expose processes' high-resolution timers through the "`/proc/`*pid*`/stat`" interface.[2]

The combination of a flexible reservation-based scheduling API and high-resolution timers allowed us to implement the CPU Broker on TimeSys Linux. The architecture of the broker is general, however, and we believe that it would be straightforward to port our current CPU Broker implementation to another RTOS (e.g., HLS/Linux [16], [17]) that provides both CPU reservations and accurate accounting.

B. Openness and Non-Invasiveness

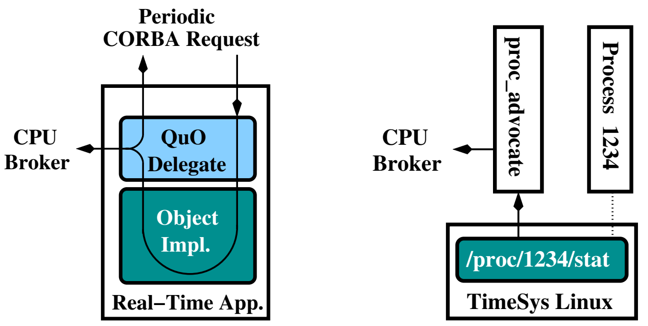
We chose to implement the CPU Broker using CORBA [1], an industry-standard middleware platform for distributed objects. CORBA provides two main features for achieving the broker's goals of openness and non-invasiveness. First, CORBA defines a standard object model and communication mechanism. By implementing the broker's advocates and policies as CORBA objects, we provide a framework for real-time systems designers to use in configuring and extending the broker. Second, CORBA abstracts over the locations of objects: communicating objects can be located in a single process, on different processes on a single machine, or on different machines. This has practical significance for both usability and performance. The broker can be easily extended with new advocates and policies without modifying existing broker code: this enables rapid prototyping, late (e.g., on-site) and dynamic customization, and cases in which a custom broker object is tightly coupled with an application (and therefore is best located in the application process). When performance is critical, new objects can be located in the same process as other broker objects; high-quality implementations of CORBA can optimize communication between colocated objects.

As described in Section I, middleware in general and CORBA in particular are increasingly important for the cost-effective development of reliable real-time and embedded systems. Using CORBA in the implementation of the CPU Broker allows us to leverage this trend. We can rely on high-quality real-time middleware — in particular, the TAO [18] real-time CORBA ORB — in our implementation and also take advantage of the increasing popularity of CORBA for the development of DRE systems. More important, however, is that CORBA provides a basis for non-invasively connecting the CPU Broker to the real-time CORBA-based tasks that it manages.

A primary goal in designing and implementing the broker was to support applications that are not developed in conjunction with our system: in other words, to support programs in which the "application logic" is decoupled from the management and control of the application's real-time behavior. This makes both the applications and our CPU Broker more flexible, and it allows real-time control to be modularized within the broker rather than being scattered throughout many programs. Effective support for this programming style requires that the broker be able to integrate with its managed tasks in a non-invasive manner, i.e., in ways that require minimal or no changes to the code of the managed tasks. The broker itself runs as a user-level process that acts as the default container for the advocate, policy, and scheduler objects described previously. Integration with real-time applications therefore requires that we build "transparent bridges" between the CPU Broker and those real-time tasks. We have implemented two strategies for non-invasive integration as illustrated in Figure 3.

The first strategy inserts a proxy object, called a *delegate*, into a CORBA-based real-time application. This strategy is appropriate when the real-time work of an application is modularized within one or more CORBA objects, as shown in Figure 3(a). The broker's delegates are implemented with QuO [4], which provides a family of languages and other infrastructure for defining and deploying delegates. The implementation of the reporting delegate class is generic and

[2]Only the Linux-standard low-resolution timers (with 10 ms granularity) are exposed in the "`/proc/`*pid*`/stat`" interface by default. TimeSys' high-resolution counters are made available through the `getrusage` system call, but that cannot be used to inspect arbitrary processes.



(a) Via QuO

(b) Via `proc_advocate`

Fig. 3. Non-invasive connections between tasks and the CPU Broker

reusable, not application-specific. Further, delegates can typically be inserted in ways that are transparent to the application, or localized to just the points where objects are created. A C++ or Java programmer might add just a few lines of code to a program's initialization or to a factory method [19]; alternatively, the code can be integrated in a non-invasive manner via aspect-oriented programming [20]. In our experience, delegates can often and effectively modularize the monitoring and control of real-time behavior in CORBA servers.

For applications that are not built on middleware, however, a second strategy is required. For these cases, we have implemented a "process advocate" (`proc_advocate`) as illustrated in Figure 3(b). The process advocate is an adapter between an unmodified application and the broker. The `proc_advocate` is a process: it requests a reservation from the broker and then forks the unmodified application as a child. The `proc_advocate`'s reservation is inherited by the child process, as described in Section IV-A. The `proc_advocate` is then responsible for monitoring the CPU usage of the application and reporting to the broker. Information about the child's resource usage is obtained from kernel's "`/proc/`*`pid`*`/stat`" interface, extended with high-resolution timers as explained previously. Other data about the child process, such as its period, are specified as command-line options to the process advocate.

C. Example Advocates and Policies

The CPU Broker implements an open framework for configurable and extensible control of real-time applications. To demonstrate the framework, in particular for tasks with dynamically changing CPU requirements, we implemented a core set of scheduling advocates and policies. These objects are generic and reusable in combination to construct a variety of adaptive systems, but we have not attempted to implement a complete toolbox. Rather, we have implemented example advocates and policies that we have found useful to date.

For conditioning the feedback from tasks with dynamically changing demands, the broker provides two advocates called *MaxDecay* and *Glacial*. MaxDecay tracks recent feedback from its task: every time it receives a report, a MaxDecay advocate requests a reservation that satisfies the application's greatest demand over its last n reports, for a configured value of n. A Glacial advocate, on the other hand, is used to adapt slowly: it adjusts the requested reservation by a configured fraction of the difference between its task's current reservation and the task's current demand. In general, MaxDecay advocates are useful for applications whose normal behavior includes frequent spikes in demand that should be anticipated, whereas Glacial advocates are useful for tasks whose spikes represent abnormal situations that are not useful in predicting future demands.[3] Other advocates provided by the broker include an auxiliary advocate for coordinating with QuO — the advocate sends the status and advice data to QuO "system condition" objects — and an advocate for logging data to files.

The broker provides three main policy objects. The first, called *Strict*, allocates reservations to tasks in order of their user-assigned importances. Requests from high-priority tasks are satisfied before those from low-priority tasks: when there is contention for CPU time, important tasks will starve less important tasks. This policy is good for periodic tasks with critical deadlines, because important tasks can be strongly isolated from less important ones. The second policy, *Weighted*, satisfies requests according to a set of user-assigned task weights. When there is contention for resources, all reservations are reduced, but in inverse proportion to the tasks' weights: "heavy"/important tasks are less degraded than "light"/unimportant tasks. This policy implements the adaptive reservation algorithm described by Abeni and Buttazzo [6], except that the inputs to our policy are possibly modified by the advocates described above. Weighted is often preferred over Strict for aperiodic and CPU-bound tasks. The third policy provided by the broker is *Partition*, which divides the available CPU resources into two or more parts. The resources within each part are controlled by a separate policy object, i.e., a Strict or Weighted policy. This allows for isolation between task groups and combinations of contention policies, e.g., for managing both periodic and aperiodic tasks. The Partition policy keeps track of which tasks belong to which groups, and it allows an administrator to move tasks between groups at run time. By manipulating the subpolicy objects, the administrator can also dynamically change the amount of CPU time available within each subpolicy.

D. Using the CPU Broker

Finally, as a practical matter, the interface to starting and configuring the CPU Broker is critical to its use. As described previously, the broker is normally run as a single process that acts as a container for the broker's CORBA objects. When this process begins, it creates a bootstrap object that can be contacted by other CORBA tools in order to configure the broker and implement a desired CPU management strategy. We

[3] Note that the distinction is based on an application's anticipated *behavior* and not its *importance*. An advocate always considers its application to be "important." It is the job of a policy, not an advocate, to make decisions based on user-assigned importance values.

provide a command-line tool, called `cbhey`, that allows interactive communication with the broker, its advocates, and its policies. For example, setting the importance of a task named `mplayer` is as simple as using `cbhey` to talk to the controlling policy object and telling that policy to "`set priority of task mplayer to 5`." Connecting an external advocate or policy object to the CPU Broker is accomplished by using `cbhey` to give the location of the object to the broker; CORBA handles communication between the broker and the external object. Tighter and automated integration with the CPU Broker is achieved by making (remote) method invocations on the CORBA objects within the broker. For example, an end-to-end QOS management framework would likely interact with the broker not via `cbhey`, but instead by making CORBA calls to the broker's objects. For performance-critical situations, we also provide a mechanism for dynamically loading shared libraries into the main broker process.

V. Evaluation

Our evaluation of the CPU Broker is divided into three parts. First, we measure the important overheads within the broker's implementation and show that they are small and acceptable for the types of real-time systems we are targeting. Second, we evaluate the broker's ability to make correct CPU allocations for a set of synthetic workloads, with and without dynamically changing resource requirements, and with and without contention. Finally, we demonstrate the CPU Broker applied to a simulated DRE military application. We extend the broker with an application-specific advocate, measure the broker's ability to make appropriate CPU allocations, and evaluate the impact on the quality of service achieved by the system in the face of CPU contention.

All of our experiments were performed in Emulab [21], a highly configurable testbed for networking and distributed systems research. Each of our test machines had an 850 MHz Pentium III processor and 512 MB of RAM. Each CPU Broker host ran TimeSys Linux/NET version 3.1.214 (which is based on the Linux 2.4.7 kernel) installed atop Red Hat Linux 7.3. The CPU Broker and other CORBA-based programs were built using TAO 1.3.6 and a version of QuO (post-3.0.11) provided to us by BBN. For experiments with the distributed military application, the three hosts were connected via three unshared 100 Mbps Ethernet links.

A. Monitoring and Scheduling Overhead

There are two primary overheads in the broker: obtaining CPU data from the kernel, and communication between the broker objects via CORBA. When an application is monitored by a QuO delegate or `proc_advocate`, communication involves inter-process communication (IPC). To measure the total overhead, we built three test applications. The first was monitored by our ordinary QuO delegate, which performs two-way IPC with the broker: it sends the task status and waits to receive new reservation data. The second was monitored by a special QuO delegate that performs one-way IPC only: it does not wait for the broker to reply. The third was monitored by an "in-broker process advocate": an advocate that functions like `proc_advocate`, but which lives in the main CPU Broker process and is run in its own thread. (The performance of an ordinary `proc_advocate` is similar to that of a two-way QuO delegate.) Each test application ran a single sleeping thread. Periodically (every 33 ms), each monitor measured the CPU usage of its task plus itself via the "`/proc/`*`pid`*`/stat`" interface and sent a report to the broker. Each test was run on an unloaded machine with a CPU Broker containing a MaxDecay advocate and a Weighted policy. We ran each test and measured the CPU and real time required for processing each of the first 1000 reports following a warm-up period.

TABLE I

Average measured overhead of reports to the CPU Broker

Configuration	Monitor+Broker CPU Time (usec)	Monitor Only Real Time (usec)
Two-way QuO delegate	1742	1587
One-way QuO delegate	1716	660
In-broker process advocate	400	400

The average times for the reports are shown in Table I. The first data column shows the average of the total user and kernel time spent in both the monitor and broker per report: this includes time for obtaining CPU data, doing IPC if needed, and updating the reservation. The second column shows the average wall-clock time required for each report as viewed from the monitoring point. This is a measure of per-report latency: in the QuO delegate cases, it is less than total CPU time because it does not account for time spent in the broker. We believe that the measured overheads are reasonable and small for the class of real-time systems based on COTS operating systems and middleware that we are targeting. Increasing the number of tasks or decreasing the tasks' periods will increase the relative overhead of the broker, but when necessary, additional optimizations can be applied. For example, one might configure the monitors with artificially large reporting periods, at some cost to the broker's ability to adapt allocations quickly.

B. Synthetic Applications

To test the broker's ability to make correct CPU reservations, we used Hourglass [22] to set up two experiments. Hourglass is a synthetic and configurable real-time application: its purpose is to analyze the behavior of schedulers and scheduling systems such as the CPU Broker.

Our first experiment tests the CPU Broker's ability to track the demands of a periodic real-time task with a dynamically changing workload. The goal is for the reservations over time to be both adequate (allowing the task to meet its deadlines) and accurate (so as not to waste resources). We assume that the system designer does not know the shape of the application's workload over time, only that it goes through phases of increased and decreased demand. In a typical case like this, a MaxDecay advocate is appropriate for adapting to the task. There is no CPU contention in this experiment, so the policy is unimportant: we arbitrarily chose to use the Strict policy.

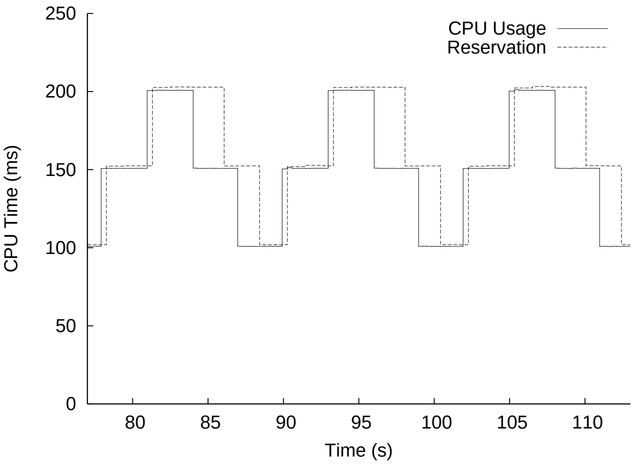


Fig. 4. Actual compute time and reserved compute time for a task with a time-varying workload. The broker is configured to adapt to increased demands immediately, and to reduced demands only after several periods of reduced need. The MaxDecay task advocate used in this experiment can be configured to track the actual demand more or less closely.

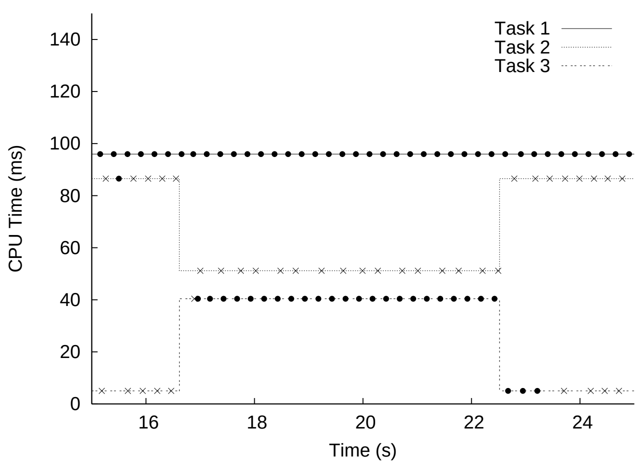


Fig. 5. Dynamically changing reservations in response to changing task importances. Task 1 requires 95 ms every 250 ms; tasks 2 and 3 require 125 ms and 40 ms with the same period. The lines show the compute time reserved for each task. When task importances change, the broker policy updates reservations according to the Strict policy. Marks on the lines show the ends of task cycles that met (•) and did not meet (×) their deadlines.

To drive this experiment, we created a test program that uses the core of Hourglass in a simple CORBA application. Our program's main loop periodically invokes a colocated CORBA object, which computes for a while and then returns to the main loop. The object's compute time can be fixed or variable in a configured pattern. We introduced a QuO delegate between the main loop and the object in order to connect our application to the CPU Broker. We configured our test application to have a period of 300 ms and to run a time-varying workload. The task goes through phases of low, medium, and high demand (with compute times of 100, 150, and 200 ms), and each phase lasts for 10 periods.

We ran our test application in conjunction with the CPU Broker, and the results are shown in Figure 4. The graph illustrates two points. First, the reservations made by the CPU Broker accurately track the demand of the application over time. Second, the MaxDecay advocate operates as intended, predicting future CPU demand based on the greatest recent demand of the task. This prevents the broker from adapting too quickly to periods of reduced demand, which is appropriate for tasks that have occasional but unpredictable periods of low activity.[4] If more accurate tracking were required, a designer would configure the advocate to observe fewer recent reports or replace the advocate altogether with a better predictor (as we do in Section V-C).

The second synthetic experiment tests the CPU Broker's ability to dynamically arbitrate CPU resources between competing tasks. There are two kinds of dynamic events that require updates to the broker's reservations: changes in the requests from tasks, and changes to the policy. Because the previous experiment concerned adaptation to tasks, we chose to focus this experiment on changes to the policy.

[4]This behavior in turn makes resource allocations more stable in multi-task systems. When there are several applications competing for resources, it is often undesirable to adjust resource allocations too frequently.

We set up three instances of the (unmodified) Hourglass application, each connected to the CPU Broker with a "continuous rate" advocate that makes a single report describing its task's desired reservation. The compute times of the tasks were set to 95, 125, and 40 ms per 250 ms period. The broker was configured with a Strict (importance-based) policy; further, the policy was set to reserve at most 75% of the CPU to all tasks.

We then ran the three Hourglass processes and used `cbhey` to change the importances of the three tasks dynamically. The results of this experiment are shown in Figure 5. At the beginning of the time shown, the importance values of the three tasks are 10, 5, and 1, respectively. The Strict policy correctly satisfies the demand of task 1, because it is most important, and this task meets its deadlines. The remaining available time is given to task 2, but this reservation is insufficient and the task generally misses its deadlines, as does task 3. (These tasks can use unreserved CPU on a best-effort basis to occasionally meet their deadlines.) At time 16.6, we raise the importance of task 3 to 7, making it more important than task 2. In response, the broker reallocates CPU resources from task 2 to task 3, and task 3 begins to meet is deadlines. At time 22.5, we lower the importance of task 3 back to 1, and the broker again recomputes its reservations.

This experiment highlights three key points. First, the configured broker policy works correctly and allocates resources to the most important tasks. As described previously, the broker implements and is open to other policies that arbitrate in different styles. Second, the broker properly and quickly adjusts reservations in response to changes to the policy configuration. In a real system, these kinds of changes could be made by an automated QOS management system or by a human operator. Third, the total utilization of the system is constantly high. In other words, the broker allocates all of the CPU time that it is allowed to assign, and the sum of the three reservations is always equal to that amount. The broker enables a system designer to keep utilization high while choosing how to divide resources in the face of contention.

C. The UAV Application

A primary goal of the CPU Broker is to provide flexible CPU control in real-time and embedded systems that are developed atop COTS middleware and that operate in highly dynamic environments. To demonstrate the benefits of the broker to these kinds of applications, we incorporated our broker into a CORBA-based DRE military simulation called the Unmanned Aerial Vehicle (UAV) Open Experimentation Platform [14]. This software simulates a system of one or more UAVs that fly over a battlefield in order to find military targets. The UAVs send images to one or more ground stations, which forward the images to endpoints such as automatic target recognition (ATR) systems. When an ATR process identifies a target, it sends an alert back to the originating UAV.

We applied our CPU Broker to the ATR stage to ensure that the ATR could reliably keep up with the flow of images coming from a UAV. The ATR task is a Java program that receives images at a rate of two frames per second. Because the ATR is CORBA-based, we used a QuO delegate to monitor the ATR and report to the CPU Broker after each image is processed. Inserting the delegate required simply adding a few lines of code to the application's main class. This change was introduced non-invasively by running the ATR with a modified class path. The Java process contains many threads, but the details of how these threads are managed by Java are unimportant to the CPU Broker. Managing the ATR relies on the broker's ability to measure the aggregate CPU consumption of all threads within a process and make a reservation that is shared by all those threads. (See Section IV-A.)

Because the ATR is a Java process, it periodically needs to garbage collect (GC). During these periods, the CPU demand of the ATR is much greater than its demand during normal periods. This kind of situation is not uncommon in software that was not carefully designed for predictable behavior. We could have dealt with this problem by configuring the CPU Broker with a MaxDecay advocate which keeps the ATR's reservation high in anticipation of future GC cycles, but this would have been unnecessarily wasteful. Instead, we implemented a custom advocate that predicts when the ATR will GC and requests an increased CPU reservation only in those cases — a *proactive*, rather than a *reactive* advocate.

The behavior of the ATR and our custom advocate are shown in Figure 6. The graph shows the periodic demand of the ATR and the predictions made for the ATR by our advocate. These lines often match closely, but our advocate is not a perfect predictor. Still, the forecast from our custom advocate is better than we could achieve with MaxDecay, which would consistently over-allocate or be slow to react to spikes.

We then ran the UAV software with and without the CPU Broker to test the broker's ability to improve the run-time behavior of the system under CPU load. We used three machines running one UAV process, one distributor, and one ATR process, respectively. The third machine also ran a simple process that receives images from the distributor and sends them to the ATR. We ran this simulation in three configurations, for 220 seconds each time, and collected data about the reliability of the system.

The results of our tests are shown in Table II. We first ran the UAV software without introducing the CPU Broker or any competing CPU loads in order to obtain baseline measures. The table shows the rate of image processing (measured at the receiver over 3-second windows), the latency of alerts (delay seen by the UAV between its sending of a target image and its receipt of the corresponding alert from the ATR), and the total numbers of images and alerts processed by the system. We then added a competing real-time task on the ATR host — an Hourglass task with a reservation for 90 ms every 100 ms — and ran the simulation again. The results in the second column show that the system behaves unreliably: many images and alerts are lost. Finally, we used the CPU Broker on the ATR host in order to prioritize the receiver and ATR processes above Hourglass. The third column shows that image handling in the broker-managed UAV system is similar to that in the system without load. Similarly, no alerts are lost, but their latencies are increased for two reasons. First, our advocate occasionally

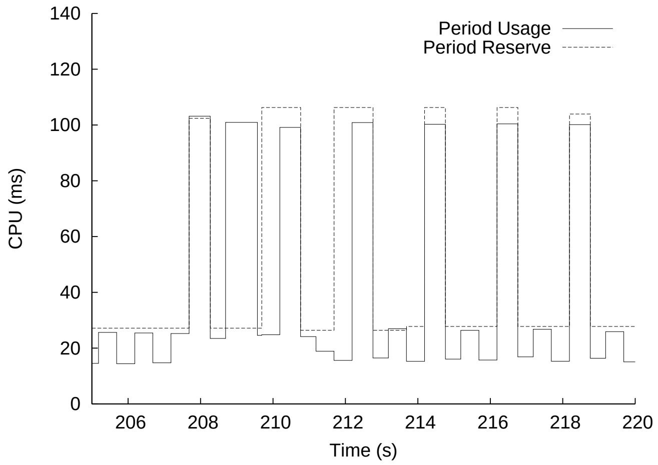


Fig. 6. Comparison of the actual compute time and reserved compute time for the ATR. The ATR's demand has regular spikes that correspond to garbage collections. The custom advocate predicts and adapts to this behavior.

TABLE II
PERFORMANCE OF THE UAV SIMULATION

Metric	Unloaded, Baseline	CPU Load	CPU Load, With Broker
Frames processed	432	320	432
Avg. frames per second	1.84	1.32	1.81
Min. frames per second	1.67	0.45	1.11
Max. frames per second	2.00	2.01	1.99
Std. Dev.	0.09	0.34	0.09
Alerts received	76	50	76
Avg. latency (ms)	127.67	1560.44	325.72
Min. latency (ms)	101.00	362.00	145.00
Max. latency (ms)	193.00	3478.00	933.00
Std. Dev.	33.46	961.62	153.60

mispredicts GC cycles: we could use a different advocate to improve reliability, at a cost in overall system utility. Second, although the ATR receives its reservation, the RTOS may spread the compute time over the full period (500 ms), thus increasing alert latency. We could address this problem in the future by adding deadline information to our CPU Broker interfaces. In sum, our experience showed that the broker can non-invasively integrate with a CORBA-based DRE system and improve that system's reliability in the face of CPU contention.

VI. Conclusion

Embedded and real-time systems are increasingly dependent on the use of COTS infrastructure and the reuse of software parts — even entire applications. Furthermore, systems are increasingly deployed in environments that have changing sets of computing resources and software with dynamically changing requirements. We have presented the design and implementation of our CPU Broker that addresses the needs of these systems in an open and extensible fashion. Our architecture supports adaptive, feedback-driven CPU reservations and explicitly separates per-task and global adaptation strategies. Our implementation atop a commercial RTOS effectively determines and adapts CPU reservations to the dynamic requirements of its managed tasks, with low overhead. Finally, the broker effectively modularizes the strategy for allocation and adaptation, and connects to both middleware-based and other applications in a non-intrusive manner. In conclusion, we believe that the broker approach can provide important benefits toward achieving understandable, predictable, and reliable real-time behavior in a growing and important class of real-time and embedded software systems.

Availability

The CPU Broker is open-source software. Complete source code and documentation for the CPU Broker are available at `http://www.cs.utah.edu/flux/alchemy/`.

Acknowledgments

We thank Craig Rodrigues at BBN for his continual and ongoing help with the UAV software. We also thank Alastair Reid for his valuable comments on drafts of this paper. Finally, we thank Leigh Stoller, Rob Ricci, Mike Hibler, Kirk Webb, and other members of the Emulab support staff for helping us deploy and evaluate the CPU Broker in that facility.

References

[1] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Dec. 2002, revision 3.0.2. OMG document formal/02–12–06.

[2] ——, *Real-Time CORBA Specification*, Nov. 2003, version 2.0. OMG document formal/03–11–01.

[3] TimeSys Corporation, "TimeSys Linux GPL: Performance advantages for embedded systems," 2003, white paper, version 1.1.

[4] J. A. Zinky, D. E. Bakken, and R. D. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55–73, 1997.

[5] J. Regehr and J. Lepreau, "The case for using middleware to manage diverse soft real-time schedulers," in *Proceedings of the International Workshop on Multimedia Middleware (M3W '01)*, Ottawa, ON, Oct. 2001, pp. 23–27.

[6] L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, China, Dec. 1999, pp. 70–77.

[7] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)*, Austin, TX, Dec. 2002, pp. 71–80.

[8] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, Vancouver, BC, June 1999, pp. 111–120.

[9] T. Nakajima, "Resource reservation for adaptive QOS mapping in Real-Time Mach," in *Parallel and Distributed Processing: 10th International IPPS/SPDP '98 Workshops*, ser. Lecture Notes in Computer Science, J. Rolim, Ed. Springer, Mar.–Apr. 1998, vol. 1388, pp. 1047–1056, in the *Joint Workshop on Parallel and Distributed Real-Time Systems*.

[10] Object Management Group, *Real-Time CORBA Specification*, Aug. 2002, version 1.1. OMG document formal/02–08–02.

[11] C. Lu, X. Wang, and C. Gill, "Feedback control real-time scheduling in ORB middleware," in *Proceedings of the Ninth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, Washington, DC, May 2003, pp. 37–48.

[12] S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A dynamic quality of service middleware agent for mediating application resource usage," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, Madrid, Spain, Dec. 1998, pp. 307–317.

[13] L. Abeni and G. Buttazzo, "Hierarchical QoS management for time sensitive applications," in *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, Taipei, Taiwan, May–June 2001, pp. 63–72.

[14] D. A. Karr, C. Rodrigues, J. P. Loyall, R. E. Schantz, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt, "Application of the QuO quality-of-service framework to a distributed video application," in *Proceedings of the Third International Symposium on Distributed Objects and Applications (DOA '01)*, Rome, Italy, Sept. 2001, pp. 299–308.

[15] R. E. Schantz, J. P. Loyall, C. Rodrigues, and D. C. Schmidt, "Controlling quality-of-service in a distributed real-time and embedded multimedia application via adaptive middleware," Jan. 2004, submitted for publication, http://www.cs.wustl.edu/~schmidt/PDF/D&T.pdf.

[16] L. Abeni, "HLS on Linux," Nov. 2002, http://hartik.sssup.it/~luca/hls/.

[17] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, London, UK, Dec. 2001, pp. 3–14.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97: Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer, June 1997, vol. 1241.

[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec. 2002, pp. 255–270.

[22] J. Regehr, "Inferring scheduling behavior with Hourglass," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002, pp. 143–156.