A Peer-to-Peer Architecture for Delivering E-Services

Vana Kalogeraki, Jim Pruyne, Aad van Moorsel
Hewlett-Packard Laboratories
Palo Alto, CA 94304
{vana,pruyne,aad}@hpl.hp.com

Abstract

Peer-to-peer architectures have been proposed to bring an earthquake to interactions on the Internet by enabling real-time direct sharing of computer resources and services. In this paper we use the peer-to-peer model to deliver e-services in a timely and reliable way. The challenge is to use the collective ability of many devices - wireless and wired - to work together to perform a task, solve a problem or complete a transaction. The proposed peer-to-peer based system is autonomous, decentralized and scalable. Our system is based on a multiple feedback loop structure that coordinates the applications and system resources in an integrated manner; monitors the behavior of the e-services transparently; and, schedules the system resources dynamically.

Keywords: peer-to-peer, e-services, real-time, scalable, decentralized systems

1 Introduction

We have now entered the next Internet evolution, the mass proliferation of e-services[1]. The Internet is evolving beyond being an infrastructure for publishing information that is displayed via browsers. Rather, the focus is about making the Internet work for the users. People are connecting to the Internet using different kinds of devices: PCs, laptops, mobile phones, PDAs. Mobility is enabled not by a single Internet-based wireless device, but rather by the collective ability of all devices - wireless and wired - working in concert to deliver rich context-specific e-services to anyone on the go. For example, e-services are delivered via the phone, pager, or virtually anything with a microchip in it. As the number and complexity of the e-services increases, modern content management techniques need to provide timely delivery, high reliability and quality of service (QoS) guarantees.

To provide end-to-end QoS and real-time guarantees to the end users the biggest challenge is how to manage the e-services and the underlying system resources in an integrated manner and to gracefully adapt to changes in the behavior of the services and the availability of the resources. This problem is difficult because, in practice, it is difficult to compute the whole schedule in advance as the starting times of the services are not always known a priori and the structure or behavior of the services can dynamically change over time. Furthermore, the global state of the distributed system is changing much faster than it can be communicated to the processors, therefore; the exact structure of the system cannot be known by a single centralized resource manager.

In this paper we propose a Decentralized Resource Management infrastructure for delivering e-services. The Resource Management System is middleware that coordinates the services and system resources in an integrated manner; monitors the behavior of the services transparently and obtains accurate resource projections; schedules the system resources dynamically over multiple pro-

[1]E-services are often also called web-services.

cessors; and reconfigures the objects in response to changing processing and networking conditions. The optimality of the management decisions depends on the accuracy of the profiling information collected, the measured data and the frequency of the measured events. The Resource Management System is structured as a Profiler and a Scheduler for each of the processors in the system. The system operates at two levels: (1) the intra-processor level where the Profiler on the processor works in concert with the local Scheduler to schedule the methods invoked by the services and (2) across multiple processors through a peer-to-peer protocol. The benefits of the Decentralized Resource Management System are multi-dimensional. It enables us:

- To increase the probability of satisfying the end-to-end QoS and soft real-time response time requirements for both new and existing services and to achieve steady flow of operation of the activities.
- To balance the load (utilization) of the processor and network resources by allocating the objects to the processors and reallocating them as necessary.
- To build scalable, decentralized and autonomous Resource Management systems to accommodate a high volume of user requests from geographically distributed and potentially heterogeneous platforms. Also, to increase the system efficiency as the failure of one of the processors does not propagate or affect other processors in the system.

The system is based on the Common Object Request Broker Architecture (CORBA) [10] which is a widely accepted standard for developing large-scale distributed applications over heterogeneous platforms. Distributed object computing middleware such as OMG's CORBA, Microsoft's Distributed Component Model (DCOM) and Sun's Java Remote Method Invocation (RMI) are very attractive because they shield software developers from low-level, tedious, and error-prone details and provide a consistent set of higher-level abstractions for developing distributed systems. Note though that CORBA is used only for the development of the e-services, e-services communicate with each other by asynchronous messages containing XML documents.

This paper is organized as follows. In Section 2 we present the peer-to-peer models and in Section 3 we describe the e-services. In Section 4 we discuss the architecture of the Decentralized Resource Management System. Section 5 presents the two-level feedback loop structure. Section 6 presents related work and Section 7 concludes the paper.

2 Peer-to-Peer Models

Peer-to-peer computing is defined as the sharing of computer resources and applications through direct exchange. Peer-to-peer computing supports the creation of a fully decentralized system by enabling direct and real-time sharing of services and information and by eliminating the need for a single centralized component. For example, the Gnutella [3] peer-to-peer model focuses on searching and discovering music files. The most distinct characteristic of peer-to-peer computing is that there is symmetric communication between the peers; each peer is both a client and a server.

Peer-to-peer models essentially create a virtual point-to-multipoint network of many peers built on top of the physical infrastructure. The peers are connected in an ad-hoc manner. A peer connects to the network of peers, by establishing a relationship with at least one peer currently on the network. Peers exchange messages to search for their neighbors or to discover information. For each search request, the peer searches its local repository for relevant matches and responds with the results. In addition, it propagates the search request to its own peers in the network.

The advantage of the peer-to-peer models is that they create a scalable, decentralized and autonomous infrastructure for searching and discovering data and services. However, the current solutions have the disadvantage that they propagate all the queries across the network (including nodes with high latencies), therefore the network can easily become a bottleneck. Routing between peers is important because it can affect the scalability of the system. These protocols were originally designed to accommodate a population of only a few thousand of users. Recent studies [1] have shown that we can employ local search strategies in power-law networks for efficient searching on the peer-to-peer network.

3 E-services

E-services are complex distributed applications that can be accessed by other applications or components over the Internet and across organizational boundaries. E-services are self-contained and modular. A complex distributed e-service can be modeled as several service tasks; each task has its own resource and timing requirements and generates a result that triggers the execution of the subsequent task. A service task is defined as a sequence of method invocations of objects distributed across multiple processors in multiple domains. E-services can be composed and deployed dynamically, therefore, their arrival time is not known a priori. These contrast with static systems where the schedules of the invoked objects are usually determined in advance and remain fixed while the tasks execute. Multiple service tasks originating from different client threads can be executed concurrently. Although tasks are triggered independently and asynchronously, they are not necessarily disjoint.

The *Travel Agent* service is an example of an e-service that is responsible for an entire travel booking. It consists of an *airline* service task that finds the appropriate airline company and flight number, a *hotel* service task that ensures accommodation even if the flight is cancelled or delayed and a *car transportation* service task that provides transportation to the user's destination. The priority with which the tasks execute is important. For example, if a passenger's flight is cancelled, the airline service task has to run in a high priority to book the next available flight to the passenger's destination.

Service tasks enable end-to-end scheduling in that they span processor boundaries and carry scheduling parameters from one processor to another yielding system-wide scheduling strategies that require only local computations. A task's scheduling parameters apply to local threads and methods invoked by the task. The scheduling parameters depend on the scheduling algorithm implemented and can be updated during the execution of the task. The task's resource requirements depend on the resource requirements of the objects invoked by the task, the current state of the objects, and the sequence of method invocations. With each task we associate:

- *Deadline*: the time interval, starting at task initiation within which the task should be completed, specified by the end user.
- *Importance*: a metric that represents the relative importance of the task, specified by the end user. The importance metric is derived from the importance of the service requested by the user and affects the order with which the tasks are executed on multiple processors.
- *Projected_latency*: the estimated amount of time required for the task to complete. This projected latency is computed as the sum of the computation times of the methods invoked by the task and the corresponding communication times.
- *Laxity*: the difference between *Deadline* and *Projected_latency*, which represents a measure of urgency of the task. As the task executes, it is scheduled according to *Laxity*, which is dynamically adjusted.
- *Mean_invocations*: the mean number of invocations of the task made by the different users.

E-services are registered in directory services to advertize their operations. The Universal Description, Discovery, and Integration Specification (UDDI)[14] is an example of a group of web-based registries that exposes information about services and their interfaces. UDDI specifies both interfaces for describing the service registries and also specifies how the UDDI registries can be operated. This allows the services to be dynamically discovered and composed into more complicated

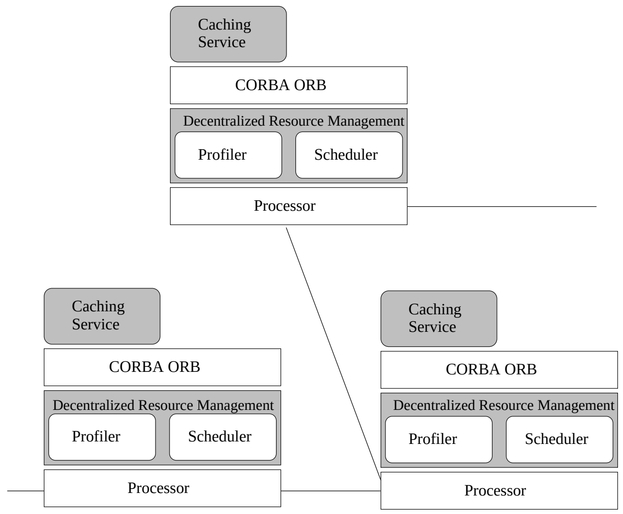

Figure 1: The Decentralized Resource Management Infrastructure.

services.

4 Decentralized Resource Management

Figure 1 shows the structure of the Decentralized Resource Management System implemented as a Profiler and a Scheduler for each of the processors in the system. The Profiler on the processor monitors the behavior of the objects on the processor and measures the current load on the processor resources. The Local Scheduler is responsible for specifying a local ordered list (schedule) for the method invocations on the processor that determines how access to the resources is granted. The Schedulers exploit information collected by the Profilers to schedule services across multiple processors to meet soft real-time deadlines.

The Profilers from different processors work in a peer-to-peer model; each Profiler collects feedback from its peers in terms of the objects on their processors and the load of their resources. The Profilers are connected in an ad-hoc basis; a direct connection is established between two Profilers only if there is a need to exchange information between these peers. For example, the Profiler computes the projected latency for the service task as the sum of the computation times of the methods invoked by the task and the corresponding communication times. In previous work [5, 6] we had developed a single (but possibly replicated and distributed) resource manager for a distributed system that has the global view of the system and is responsible for distributing the objects on the processors. Our proposed decentralized architecture is novel because it creates a Decentralized Resource Management System constructed from independent and autonomous components that cooperate in a purely decentralized world, there is no centralized component to manage all the service tasks and the resources for the whole system.

4.1 Resource Monitoring

To accommodate a variety of systems with different capabilities (from large-scale enterprise systems to small-scale highly critical systems), a Profiler on each processor measures the maximum and current usage on the processor's resource which determines the further allocation of the services in the system. A processor is characterized by its speed, the size of its local memory, and the size of its disk space. A communication link is characterized by the bandwidth of the link. Thus, each Profiler measures:

- *Load*: the current load on the processor.
- *Memory*: the memory in use on the processor.
- *Disk*: the disk space in use on the processor and the number of disk accesses.
- *Bandwidth*: the bandwidth in use on the communication links connecting the Profiler with its peers.

One of the great challenges to providing high-performance services to the end users is to address the bottleneck at the user side and the service provider side. The speed with which the users access the services in the system is limited by the capacity of their modem connections to the ISPs or their corporate connections to the Internet. The bottleneck at the service providers side is a function of the volume of user requests and the distance between the service provider and the end users.

4.2 E-service Profiling

The Profilers on the processors capture the behavior of the services by monitoring the messages exchanged between the services. Each method invocation or response, as monitored by the Profilers, is characterized by the following tuple: *(Action, local_object, invoking_method, remote_object, invoked_method, Invocation_time)*

where *Action* is determined by the Profilers [6] and is one of the following: LOCAL_START, LOCAL_COMPLETE, REMOTE_START, REMOTE_COMPLETE. The Profilers distinguish between a remote method invoking a local method on a local object (LOCAL_START, LOCAL_COMPLETE) and a local method invoking a remote method (REMOTE_START, REMOTE_COMPLETE) on a remote object, and also between the corresponding responses. The Profilers attach a timestamp to each of the method invocations and can therefore measure the execution and computation times of the local and remote methods as invoked by the tasks.

The Profiler measures the $Projected\_latency_t$ for a service task t as the sum of the computation times of the methods invoked by the task and the corresponding communication times. For each method m invoked by the task, the Profiler measures the $Mean\_computation\_time_m$ which is the mean time for the method to execute locally on the processor, including queueing time but excluding time for embedded method invocations. When a method m makes a remote method invocation to a method n of an object j on a remote processor, the Profiler records the time of invocation and updates the number of incoming and outgoing method invocations for the corresponding objects. When the invoked method n completes and returns its response to the invoking method m, the Profiler calculates the mean time for the remote invocation, $Mean\_remote\_time_{mn}$, from the time at which method m invoked method n to the time at which it received the response. The Profiler computes the $Mean\_communication\_time_{mn}$ to communicate an invocation from method m to method n and receive a response back as the difference between $Mean\_remote\_time_{mn}$ and

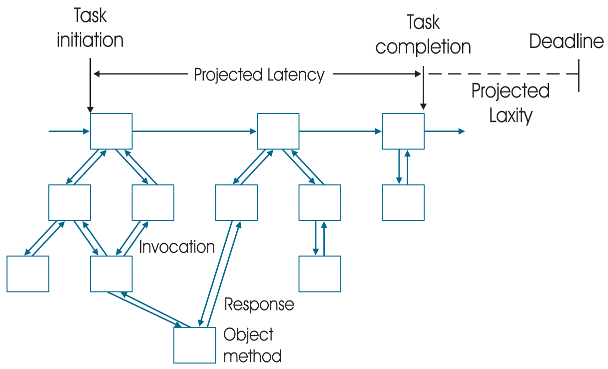

Figure 2: The Service Execution Graph for a Service.

$Mean\_local\_time_n$, where $Mean\_local\_time_n$ is the time for method n to complete the execution.

As the service executes, the methods invoked by each service task are recorded dynamically to construct the Service Execution Graph (Figure 2), that describes the flow of operation for the service. Each node in the graph represents a method m, while each edge corresponds to an invocation of method m or a response. The Service Execution Graph also describes the relationships between the processors as the service tasks invoke methods on objects across multiple processors and domains.

Maintaining accurate profiling information is important for two reasons. First, it is the means by which the Profiler can detect an external load and act upon it. The Profiler can identify an object that is causing a large queueing delay for a task and, therefore, can detect significant deviations in performance. Second, it allows the Profiler to assemble a picture of the system by collecting feedback from its peers. It is important to note though, that no Profiler has a global view of the system or a complete view of a Service Execution Graph. Every graph defines a subset of processors. These are the processors that execute tasks for this execution graph. The Profiler obtains information only from those peers that service tasks from the same execution graph. The Service Execution Graph for a service is constructed by aggregating the views of the corresponding Profilers from the graph. This is the second level feedback loop structure of the Decentralized Resource Management Domain that uses measurements of elapsed time and measurements of processor loads to refine the initial estimates of the laxity of the tasks as they start. The second level feedback loop operates on a second-by-second basis.

4.3 Distributed Scheduling

The challenge in a distributed system is to provide predictability of timeliness over multiple processors when the objects are invoked concurrently and asynchronously by multiple service tasks and compete for limited computing resources. For multi-processor environments, it has been shown [2] that no scheduling algorithm is optimal without *a priori* knowledge of the deadlines, computation times and arrival times of the tasks. In practice, however, it is impractical to compute the whole schedule beforehand because the services are constructed and deployed dynamically. Also, worst-case allocations are usually not effective, because they trade resource utilization for accurate predictions and can result in underutilized processors. In such environments, dynamic scheduling algorithms are more applicable and flexible than static scheduling algorithms to provide the timeliness guarantees to the service tasks.

For each service task t, the local Scheduler computes the initial laxity of the task as:

$$Laxity_t = Deadline_t - Projected\_latency_t$$

where $Deadline_t$ is the time within which the task should be completed and $Projected\_latency_t$ is the estimated time to task completion. The laxity value of a task represents a measure of urgency for the task. The task's initial laxity is computed based on information collected by the Profilers during previous executions of the task. If no such information is available, the Profiler estimates the computation time for the task as a proportion of the task's deadline. This information is stored in the Service Execution Graph, kept by the Profiler, along with other information about the task. As the task executes, the methods of the objects invoked by the task are scheduled according to the remaining laxity of the task. The laxity value is updated based on the estimated computation time of the methods of the objects and their actual time measured by the Profilers during the executions. The Local Scheduler on each processor subtracts from the remaining laxity $Laxity_t$, the difference between the actual time $Computation\_time_m$ for executing method m measured by the Profilers and the $Mean\_computation\_time_m$ calculated by the Profiler based on previous executions of method m. In [7] we discuss the optimality of least laxity scheduling compared to earliest deadline first scheduling, due to the fact that least laxity scheduling considers the computation times of the tasks to derive the laxity values.

When a service task invokes a method on a remote processor, it carries with it the caller's scheduling parameters (laxity value) included in the message header. As the execution of the service moves from processor to processor, its scheduling needs are carried along and honored by the scheduler on each processor. When the service task returns, the scheduling parameters are propagated back to the caller. When the Reply is received, the actual time required is compared with the Projected time and the difference is used to adjust the task's laxity value.

The adjustment of the laxity, $Laxity_t$, provides the first level of the feedback loop structure of the Decentralized Resource Management System. If the invocation completes more quickly than was projected, the task laxity increases and the task's scheduling priority decreases. If the invocation completes more slowly, the task laxity decreases and the task's scheduling priority increases. All computations are simple and local, allowing the loop to operate on a hundreds of millisecond basis. In contrast, the feedback loops used to estimate the projected task latency, from which the initial task laxity is derived, operate more slowly on a timescale of seconds.

4.4 Service Caching

The challenge in the decentralized system is how to distribute the services on the processors to maximize the probability that the end-to-end real-time response requirements of the end users are met, given the dependencies among the services and the resource requirement constraints. Service distribution is important, because the execution times of the service tasks are affected by the number of tasks in the system, the objects invoked by the tasks, and the processing and communication times of the methods of the objects invoked by the tasks. Note though that these factors are not always compatible. For example, load balancing requires distributing the objects, while performance requires to collocate them. Furthermore, equally distributing the load on the processors can cause potential resource fragmentation on the processors.

We propose a service caching scheme that is aimed to improve performance and availability by replicating the services requested by the users and storing them on user locations. The advantage is that the services are distributed in service caches closer to the end users instead of being delivering from centralized services via long distance communication links. Essentially, we are building a network of replicated content caches, each positioned as close as possible to the users that are located on the edges of the network. Content is both originated and delivered from the edges of the network, allowing the system to balance the load across multiple resources and to accommodate a high volume of user requests from geographically distributed and potentially heterogeneous platforms. Our work is different from Content Delivery and Distribution services such as Akamai which maintain a number of centralized servers to store the data delivered to the users.

Even the simpler problem of finding an optimal deployment of the objects for a single service that invokes methods on objects across multiple processors is NP-hard. The problem becomes more complicated as user requests arrive dynamically; their arrival times is not know `a priori`; they concurrently and asynchronously invoke the same services; and, they compete for shared computing resources. Consequently, the cost of finding an exact solution is unjustified. Furthermore, the simplest scheme, to make all the services available locally for the users is not very efficient, as it has the disadvantage that all the popular services will be highly replicated, and often distributed in the same networks and, thus, will increase the cost of maintaining consistency among all the service replicas. To determine the best location to deploy a service, we consider the number of service copies in the network caches, the load on the resources, the latencies of the services and the number and frequency of invocations made by the users. Our algorithm uses current resource measurements and service profiles constructed during the previous executions of the tasks to distribute the services in a way that (1) minimizes the network traffic on the communication links and (2) balances evenly the load on the processors' resources. Service distribution uses measurements of processor loads and measurements of service laxities to decide the allocation of the service tasks on the processors. This loop operates on a multiple second basis.

5 Multiple Feedback Loops

Our Decentralized Resource Management System works in two levels: the intra-processor level and the inter-processor level.

- The inter-processor level uses a peer-to-peer distributed architecture that is (1) scalable as it balances the load across multiple resources and domains and (2) autonomous as it makes dynamic decisions in response to changes in processing and networking conditions in the system.
- The intra-processor level uses a multiple feedback loop structure that employs a least laxity scheduling algorithm that schedules the methods invoked by the tasks over milliseconds and a profiling algorithm that monitors the behavior of the objects and the usage of the resources over seconds.

The Decentralized Resource Management System (Figure 3) operates on a scheme that is adaptive. As the services execute, a profile of the method invocations for each service task is constructed dynamically and is added to the Profilers' repository. The Profilers use this information for the subsequent service invocations. When a new

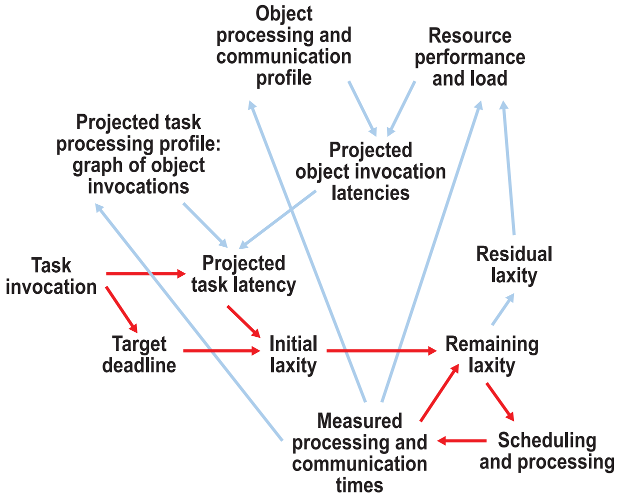

Figure 3: The Scheduling and Profiling Loop of the Decentralized Resource Management System.

service is deployed, a tentative allocation of the new objects on the processors is performed using a greedy algorithm that tries to minimize the network bandwidth on the communication links and distribute evenly the load among the processors. This increases the load on the processors and requires new projections of the task latencies. As the number of user requests increases or the latency of a service becomes too high, our service caching scheme improves the performance and availability provided to the end users by creating a new service replica closer to the edges where the requesting users are located.

6 Related Work

Many researchers [11] have realized the need for systems that can adapt to dynamic, unpredictable changes in the computing environment. Nett *et al* [9] have developed an adaptive object-oriented system using integrated monitoring, dynamic execution time prediction and scheduling to provide time-awareness for standard CORBA object invocations. Sydir *et al* [13] have implemented an end-to-end QoS-driven resource management scheme within a CORBA-compliant ORB, called ERDoS. They provide end-to-end QoS requirements corresponding to the resource demand requirements of each individual object and use an information-driven resource manager that enables applications to achieve their QoS requirements.

Research into scheduling has been dominated by hard real-time systems, but some useful results are available for soft real-time distributed systems. Jensen *et al* [4] propose soft real-time scheduling algorithms based on application benefit, obtained by scheduling the applications at various times with respect to their deadlines. Their goal is to schedule the applications so as to maximize the overall system benefit. Stankovic *et al* [12] discuss the Spring Kernel developed for large complex real-time systems. They classify the tasks based on their importance and timing requirements and use value-based functions to drive the schedule.

Nahrstedt *et al* [8] have employed resource management mechanisms to provide end-to-end QoS guarantees for multimedia computing and communication. They present a soft real-time scheduler for the Unix environment and a resource broker that provides QoS, negotiation, admission and reservation capabilities for sharing resources, such as memory and CPU. Their dynamic scheduler is based on a preliminary round of testing to capture the behavior of the tasks before the actual execution starts.

7 Conclusions

The increasing need to share resources and information, the decreasing cost of powerful workstations, the widespread use of networks and the maturity of software technologies will further increase the use of distributed systems and applications and so too the demand for more efficient resource management. We have proposed a scalable and autonomous decentralized resource management architecture structured as a Profiler and a Scheduler for each of the processors in the system. The Decentralized Resource Management System operates at two levels: within the processor by collecting feedback from the local tasks and resources and across the processors through a peer-to-peer protocol. The system allows activities with different levels of temporal granularity, scheduling at the level of milliseconds, profiling over seconds, and load balancing at the level of multiple seconds.

References

[1] L. A. Adamic, R. M. Lukose, A. R. Puniyani and B. A. Huberman, "Search in power-law networks", manuscript.

[2] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12 (December 1989), pp. 1497-1506.

[3] Gnutella, http://www.gnutella.wego.com/

[4] E. D. Jensen, C. D. Locke and H. Tokuda, "A time-driven scheduling model for real-time operating systems," *Proceedings of the IEEE 6th Real-Time Systems Symposium*, San Diego, CA (December 1985), pp. 112-122.

[5] V. Kalogeraki, P.M. Melliar-Smith and L.E. Moser, "Dynamic migration algorithms for distributed object systems," *Proceedings of the IEEE 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona (April 2001), pp. 119-126.

[6] V. Kalogeraki, P.M. Melliar-Smith and L.E. Moser, "Using multiple feedback loops for object profiling, scheduling and migration in soft real-time distributed object systems," *Proceedings of the IEEE Second International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint Malo, France (May 1999), pp. 291-300.

[7] V. Kalogeraki, P. M. Melliar-Smith and L. E. Moser, "Dynamic scheduling of distributed method invocations," *Proceedings of the IEEE 21st Real-Time Systems Symposium*, Orlando, Florida (November 2000), pp. 57-66.

[8] K. Nahrstedt and R. Steinmetz, "Resource management in networked multimedia systems," *Computer*, vol. 28, no. 5 (May 1995), pp. 52-63.

[9] E. Nett and M. Gergeleit and M. Mock, "An adaptive approach to object-oriented real-time computing," *Proceedings of the IEEE 1st International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan, (April 1998), pp. 342-349.

[10] Object Management Group, *The Common Object Request Broker Architecture*, formal/99-10-07, Version 2.3.1, October 1999.

[11] S. Saewong and R. Rajkumar, "Cooperative scheduling of multiple resources," *Proceedings of the IEEE 20th Real-Time Systems Symposium*, Phoenix, AZ (December 1999), pp. 90-101.

[12] J. A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating system," *Operating Systems Review*, vol. 23, no. 3 (July 1989), pp. 54-71.

[13] J. J. Sydir, S. Chatterjee, and B. Shabata, "Providing end-to-end QoS assurances in a CORBA-based system," *Proceedings of the IEEE First International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan (April 1998), pp. 53-61.

[14] The Universal Description, Discovery, and Integration Specification, http://www.uddi.org