Decentralized Resource Management for Real-Time Object-Oriented Dependable Systems

Vana Kalogeraki,
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-93
April 12th , 2001*

E-mail: vana@hpl.hp.com

real time, decentralized resource management, distributed systems, load balancing

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 have increased the demand for more efficient resource management. In this paper we present a decentralized Resource Management System based on the peer-to-peer computing model. Our system coordinates the applications and system resources in an integrated manner; monitors the behavior of the applications transparently and obtains accurate resource projections, schedules the system resources dynamically over multiple machines and reconfigures the objects in response to changing processing and networking conditions. The architecture is based on a multiple feedback loop structure that uses measurements of elapsed time and measurements of processor loads to refine the initial estimates and revise the allocation of the objects on the processors.

Decentralized Resource Management for Real-Time Object-Oriented Dependable Systems

Vana Kalogeraki
Hewlett-Packard Laboratories
Palo Alto, CA 94304
vana@hpl.hp.com

Abstract

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 have increased the demand for more efficient resource management. In this paper we present a decentralized Resource Management System based on the peer-to-peer computing model. Our system coordinates the applications and system resources in an integrated manner; monitors the behavior of the applications transparently and obtains accurate resource projections, schedules the system resources dynamically over multiple machines and reconfigures the objects in response to changing processing and networking conditions. The architecture is based on a multiple feedback loop structure that uses measurements of elapsed time and measurements of processor loads to refine the initial estimates and revise the allocation of the objects on the processors.

1 Introduction

The proliferation of powerful workstations and advances in the communication technologies facilitate the development of computer applications as real-time distributed systems. Yet, the increase in complexity, capability and size of real-time distributed systems has increased the difficulty in managing the applications and the system resources. For example, in real-time distributed systems, activities invoke methods on objects across multiple processors and have end-to-end quality-of-service (QoS) and real-time requirements. Managing these activities efficiently to guarantee that the end-to-end QoS requirements are met is a difficult task. The problem is that a single resource manager component does not suffice to control and manage efficiently the access to the resources of the distributed system.

In a real-time distributed system, where activities invoke methods on objects across multiple processors, local scheduling of the applications does not suffice to ensure that real-time deadlines will be met; rather, a system-wide scheduling strategy must be employed. In practice, it is difficult to compute the whole schedule in advance as the starting times of the applications is not always known a priori and the structure or behavior of the applications can change dynamically. Furthermore, in a distributed system, the global state of the system is changing must faster than it can be communicated to the processors, therefore; the exact structure of the system cannot be known by a resource manager. The optimality of the allocation decisions depends on the accuracy of the information collected, the measured data and the frequency of the measured events. This information may be inaccurate and incomplete. Furthermore, even an optimal initial distribution may not suffice because, when many objects are invoked concurrently by many activities, the processing power and available bandwidth can easily become a bottleneck. The Resource Manager must dynamically allocate and reallocate the application objects to balance the load on the resources over changing processing and network conditions.

In this paper we present a decentralized Resource Management System that coordinates the applications and system resources in an integrated manner; monitors the behavior of the applications transparently and obtains accurate resource projections, schedules the system resources dynamically over multiple machines and reconfigures the objects in response to changing processing and networking conditions. Managing the applications efficiently is a key requirement to support a variety of applications with different needs and to gracefully adapt to unpredictable application behavior and to changes in the load and in the availability of the resources. The benefits of the distributed Resource Management System are multi-dimensional. The Resource Management System enables us:

- To increase the probability of satisfying soft real-time response time requirements for each activity and to achieve steady flow of operation of the activities
- To balance the load (utilization) of the processor and network resources by allocating the application objects to the processors and reallocating them as necessary
- To improve the system scalability by distributing the load over multiple Resource Managers and multiple domains and creating new Resource Managers dynamically to manage a large volume of application objects efficiently

The system is based on the Common Object Request Broker Architecture (CORBA) [11] 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.

This paper is organized as follows. In Section 2 we present an overview of the distributed Resource Management System. Sections 3 and 4 present the system model and metrics. In Section 5 we describe how to manage the resource management domain and in Section 6 we describe our distributed real-time scheduling algorithm. In Section 7 we explain how we determine the number of Resource Manager Peers. Section 8 presents the multiple feedback loop structure. Section 9 presents related work and Section 10 concludes the paper.

2 The Resource Management System in a Peer-to-Peer Model

The Resource Management System is structured as a set of Resource Management Domains. A Resource Management Domain is defined as a set of nodes with the same resource management policies (e.g., the same scheduling policy). The resource management domain typically contains several processors, and a processor may belong to several resource management domains. An application may consist of a number of application tasks that can span multiple resource management domains.

Each Resource Management Domain (Figure 1) is structured as a single Resource Manager for the domain and Profilers and Scheduler for each of the processors in the domain. The Resource Manager is implemented as a collection of CORBA objects that are allocated to various processors across the domain and that are possibly distributed and replicated to increase reliability; logically, however, there is only a single copy of the Resource Manager. The Resource Manager works in

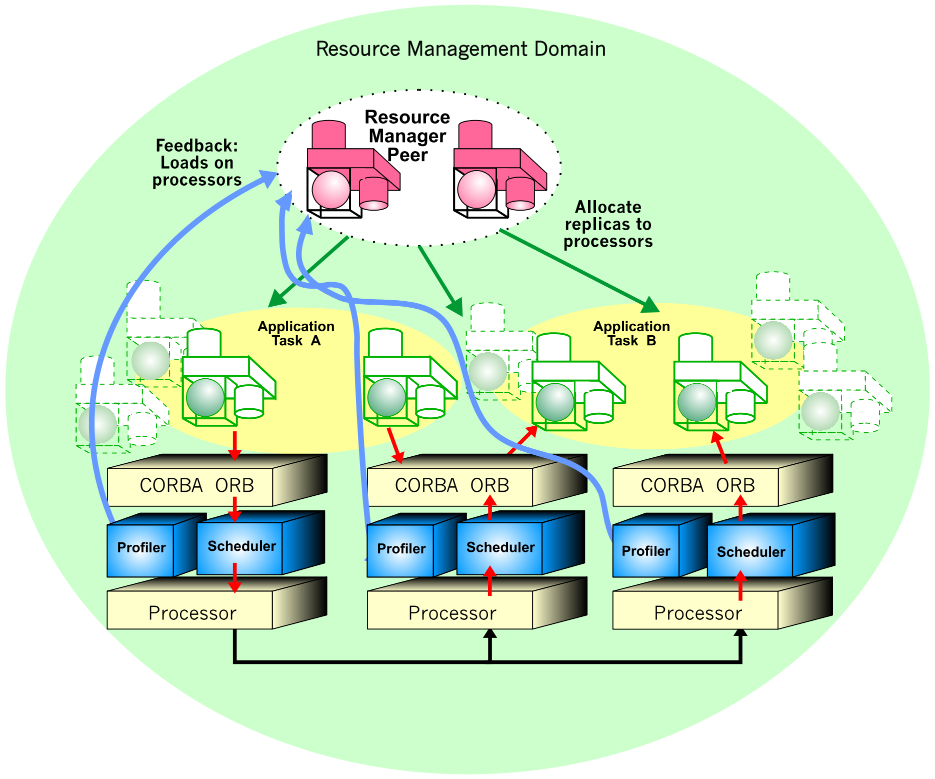


Figure 1: The Resource Management Domain.

concert with the Profilers and the Schedulers in the domain. Profilers and Schedulers that belong to more than one domains, work together with each of the Resource Managers in the domains.

The Profiler and the Scheduler on each processor are implemented in a layer between the CORBA ORB and the operating system. The Profiler on the processor monitors the behavior of the application 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 Resource Manager to schedule activities across multiple processors to meet soft real-time deadlines.

The Resource Managers in the domains communicate with each other in a decentralized, peer-to-peer computing model. The most important feature of peer-to-peer computing is that there is symmetric communication between the peers; each peer is both a client and a server and has the same capabilities and responsibilities. The advantages of a peer-to-peer computing are multi-dimensional: (1) increases the system scalability, as it distributes the management load among multiple Resource Managers and at the same time enables the applications to invoke methods on objects across different domains; (2) increases the system dependability as the failure of one of the Resource Management domains does not propagate to other domains; (3) builds an autonomous resource management domain whose management decisions affect only the applications and objects in the domain; (4) enables interoperability across geographically distributed and potentially heterogeneous object-oriented architectures.

The number of the Resource Management Domains is determined dynamically and depends on the number of application tasks and objects in the domain. Typically, a single resource management domain contains a large number of applications and, therefore, a large number of objects. The Resource Manager in each domain maintains a repository with the current view of the application objects in the domain and the current load on the system resources by collecting feedback from the Profilers. The Resource Manager obtains a global view of the system by receiving feedback information from its Resource Manager peers.

The behavior of the Resource Managers among the peers is defined through the CORBA Interface Definition Language (IDL) interface that describes the type signatures of the operations that the Resource Manager embodies. The advantage is that it separates the description of the Resource Manager behavior from its actual implementation, independently of the programming language, operating system or communication medium. The Resource Managers communicate through the exchange of messages. The CORBA Object Request Broker (ORB) packages the method invocations and responses into messages formatted according to the Internet Inter-ORB Protocol (IIOP) and subsequently passes these messages to TCP/IP for communication to the other Resource Manager components over the same ORB or over different ORBs.

3 System Model

A complex distributed application can be modeled as several application tasks; each application task has its own resource and timing requirements and generates a result that can trigger the execution of a subsequent task.

An application task is modeled as a sequence of method invocations of objects distributed across multiple processors in multiple domains. The execution of a task is triggered by a client thread (as the result of a timer signal or the completion of another operation) from a single domain and completes when the client thread finishes execution. A task is executed by a single thread or multiple threads executing in sequence or in parallel on one or more processors. Multiple tasks originating from different client threads can be executed concurrently.

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.

Each task is monitored until the client thread completes. Tasks invoke methods on objects, and objects are scheduled based on the scheduling parameters carried along with the tasks. Although tasks are triggered independently, they are not necessarily disjoint. Thus, different tasks can concurrently invoke methods on the same objects with the same or different scheduling parameters. As objects are invoked concurrently and asynchronously on the processors, they compete for the same limited processing and computing resources.

The method invocations for each task are recorded and stored in the Method Invocation Graph that describes the relationships among the objects and represents the flow of operation for the task. Figure 2 shows an example of a Method Invocation Graph. Each node in the graph represents a method, while each edge corresponds to an invocation of the method or a response. The Method Invocation Graphs for the different tasks have different roots but are not necessarily disjoint.

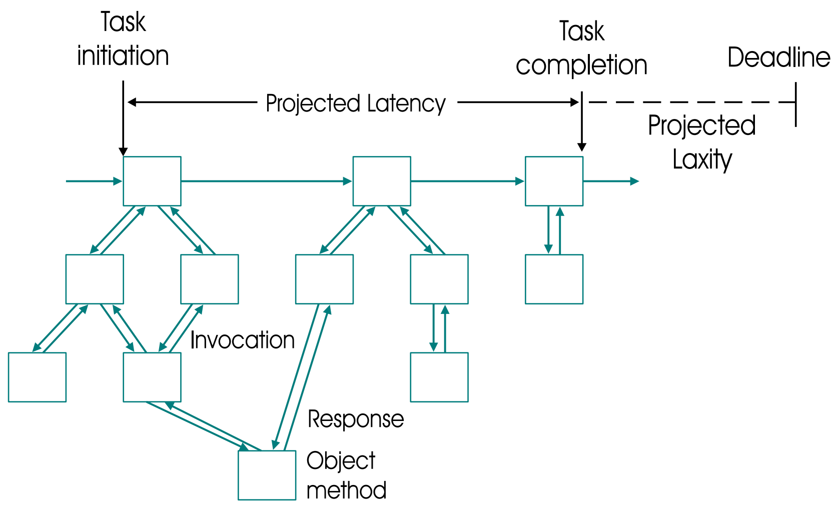


Figure 2: Method Invocation Graph.

4 System Metrics

4.1 Task Metrics

Tasks are aperiodic and 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. Each task t is associated with the following parameters:

- $Task\_identity_t$: each task has its own identity. A task starts when a client thread is invoked and finishes when the client thread completes execution. Multiple client threads can be executing concurrently, each of them characterized by a different id.

- $Task\_state_t$: describes the current state of the task, that is, whether the task is idle, running or waiting for a resource. Typical task states are: `RUNNING, WAITING and INACTIVE`.

- $Deadline_t$: the time interval, starting at task initiation within which the task t should be completed, specified by the application designer.

- $Importance_t$: a metric that represents the relative importance of the task t, specified by the end user. The importance metric affects the order with which the tasks will be executed on the processors.

- $Projected\_latency_t$: the estimated amount of time required for the task to complete. This time is measured by run-time monitoring functions and includes queueing time. Thus, the $Projected\_latency_t$ depends on all the running tasks in the system and the processors where they execute.

- $Laxity_t$: the difference between $Deadline_t$ and $Projected\_latency_t$, a measure of urgency of task t. The objects invoked by the task t are scheduled according to $Laxity_t$, which is dynamically adjusted as the task executes. The remaining laxity when the activity t completes is recorded as $Residual\_laxity_t$.

4.2 Object Metrics

For each object i, the Resource Manager maintains:

- $Methods_i$: the set of methods of object i
- $Processor\_name_i$: the processor where the object i is located
- $Object\_importance_i$: the importance of object i to each task that invokes it. If more than one task invokes the object, the object can have a different importance value to each of the tasks.
- $Average\_response\_time_i$: the average response time for object i.
- $Incoming\_invocations_i$: the number of objects that invoke methods on object i.
- $Outgoing\_invocations_i$: the number of objects whose methods are invoked by object i.
- $CPU\_utilization_i$: the percentage of the processing load required for executing methods of the object.
- $Network\_bandwidth_i$: the bandwidth on the communication links required for object's i incoming and outgoing invocations.
- $Memory\_bandwidth_i$: the memory in use for object i.
- $Fault\_monitoring\_interval_i$: the time interval between successive `pinging` operations of the object to detect if the object is still alive.

4.3 Method Invocation Metrics

For each method m, the Resource Manager maintains:

- $Method\_Name_m$: the name of the method
- $Object_m$: the object of which m is a method
- $Mean\_local\_time_m$: the mean time required, after receipt by a processor of a message invoking method m, for that processor to complete the invocation. This excludes communication time but includes queueing time and the time of embedded invocations of other methods
- $Mean\_execution\_time_m$: the mean time required, after receipt by a processor of a message invoking method m, for the method to execute locally on the processor. The $Mean\_execution\_time_m$ includes queueing time but excludes the time of invocations of other methods.
- $Mean\_remote\_time_{mn}$: the mean time required for method m to invoke method n remotely, including communication and queueing time and also the time of embedded invocations of other methods.
- $Mean\_communication\_time_{mn}$: the mean time to communicate an invocation from method m to method n and to communicate the response back. The $Mean\_communication\_time_{mn}$ is computed as the difference between $Mean\_remote\_time_{mn}$ and $Mean\_local\_time_n$.

- $Mean\_processing\_time_{mp}$ or (τ_{mp}): the mean time required for method m to execute on processor p, excluding queueing time and the time required for embedded invocations of other methods.

- $Mean\_invocations_{mn}$: the mean number of invocations that a single invocation of method m makes on method n.

5 Managing the Domain

The Resource Manager for a domain is responsible for managing the application objects of the domain and monitoring the processor and network resources of the domain. The Resource Manager records the location of the application objects on the nodes and the interactions among the objects as they are invoked by the tasks and reported by the Profilers on the processors.

5.1 Profiling

The Profiler on each procesor measures the current utilization of the processor's resources (CPU, memory, disk) and the bandwidth in use on the communication links among the processors. The Profiler also monitors local and remote method invocations and measures the actual computation and execution times of the methods of the objects. For each method m of an object i invoked locally on the processor, the Profiler measures the usage on the processor's resources during the method execution.

Each method invocation or response, as monitored by the Profilers, is characterized by the following tuple:

$$< Action, local\_object\ i, invoking\_method\ n, remote\_object\ j, invoked\_method\ m, Invoc\_time\ T >$$

where *Action* is determined by the Profilers [7] 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.

Maintaining accurate profiling information is important for two reasons. First, it is the means by which the Resource Manager can detect an external load and act upon it. The Resource Manager 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 Resource Manager to have a global view of the system in terms of which are the most appropriate processors to host new objects. Given the queueing latency of the objects on the processors, the Resource Manager can identify an overloaded processor as the processor, on which the allocated objects on average experience a large delay in the latency to the task completion.

For each object i, the Profiler measures the $Mean\_computation\_time_m$ of all the methods m of the object i and the $Mean\_number_{nm}$ of invocations per method m. The Profiler also measures the time the object spends on the Scheduler's local queue and computes the $Mean\_response\_time_i$ of the object on the processor. When a method m of object i makes a remote method invocation to a method n of an object j, the Profiler records the time of invocation, updates the number of incoming and outgoing invocations for objects i and j respectively. When the invoked method n completes and returns its response to the invoking method m, the Profiler calculates an actual time for the remote invocation, $Remote\_time_{mn}$, from the time at which method m invoked method n to the time at which it received the response. The Profiler averages this time into $Mean\_remote\_time_{mn}$. 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 $Mean\_local\_time_n$, where $Mean\_local\_time_n$ is the time for method n to complete the execution.

The Profilers use a smoothing function based on exponentially weighted averaging to compute the mean values of the methods' computation and communication times and keep a history with the method execution patterns on the processor to use for subsequent executions. Thus, the Profilers computes a local mean of the method computation and communication times on the processor and the Resource Manager computes a global mean across multiple processors. For each method m invoked locally on the processor, the Profiler supplies feedback information to the Resource Manager to determine the further allocations and reallocations of the objects on the processors. The time interval between successive Profiler reports is dynamically adjusted based on the variability of the applications and the flow of operation in the system. Our previous measurements [7] indicate that careful selection of the monitoring interval is required, because the Resource Manager uses load fluctuations reported by the Profilers to adjust the allocation of the objects to the processors.

5.2 Object Allocation

The challenge in a distributed system is to guarantee that the tasks will meet their deadlines, given object dependencies and resource requirement constraints.

We represent the network topology of the resource management domain with a graph $G = (V, E)$ where the nodes are the processors and the edges are the communication links. Each processor is characterized by the 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. Each of these resources have a maximum and a current value measured by the Profilers at runtime.

For each remote method invocation or response, the Profiler measures the bandwidth in use on the communication links between the processors where the invoking and invoked objects are located. The Profiler computes the total bandwidth used on the links connecting the processors as the sum of the bandwidth used for all the object invocations between these two processors. Note here, that, that the Profiler distinguishes between objects collocated on the same processor when there is no need to measure the network bandwidth.

For each new application task t that arrives in the domain, the Resource Manager tries to distribute the objects on the processors to maximize the probability that the end-to-end real-time response requirements of the application task are satisfied. Load balancing is important, because the execution times of the 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. For example, the sum of the available CPU capacity in the system can be sufficient to accommodate an object with a high load, but there may not be enough available CPU capacity in a single processor for such an object.

However, even the simpler problem of finding an optimal deployment of the objects for a single application task that invokes methods on objects across multiple processors is NP-hard. The problem becomes more complicated as application tasks arrive dynamically; their arrival times is not know `a priori`; they concurrently and asynchronously invoke the same objects; and, they compete for shared computing resources. Consequently, the cost of finding an exact solution is unjustified.

We propose a greedy algorithm that attempts to distribute the application objects in a way that (1) minimizes the network traffic on the communication links and (2) distributes evenly the load on the processors. The Resource Manager uses knowledge obtained from the Profilers during the previous executions of the task to estimate the amount of processor and communication resources required for executing the objects of the task and deploying the objects on the processors. The greedy algorithm is employed at run-time and ensures that the available capacity of each communication channel and processor is not exceeded. The system configuration can change dynamically in response to changes in resource loads.

5.3 The Greedy Placement Algorithm

Let $Link\_capacity_{pq}$ be the capacity of the communication link connecting processors p and q, and $Max\_bandwidth_{pq}$ be the bandwidth currently in use on the link, measured by the Profilers. The Resource Manager estimates the available amount of bandwidth on the communication link as:

$$Avail\_bandwidth(E_{pq}) = Link\_capacity(E_{pq}) - Max\_bandwidth(E_{pq})$$

The currently available bandwidth on the communication link is a dynamic quantity, that changes as the estimation of the network traffic changes. A more sophisticated monitoring algorithm could also detect periodic patterns of utilization.

Round-robin load balancing methods create unequal, highly variable, and unpredictable load distribution among the objects. Therefore, the Resource Manager uses lower `low` and upper `high` bounds to represent the utilization of the CPU on each processor. The Resource Manager identifies an underloaded processor whose current load is below `low` as the best candidate to host a new object. A processor whose load exceeds load `high` is considered an overloaded processor and should not host any more objects. The Resource Manager uses the Profilers' feedback to compute these bounds dynamically based on the flow of the operations in the system. For example, a large number of objects with low load will require a larger value for the lower `low` bound, while objects with high load, will require a lower value for the bound.

To find the paths of smallest weight between an invoking and invoked object, we use a link weighting distance function based on the number of network links that connect the two objects. The link weighting function gives high weight to the links that are used very heavily, such as:

$$w(E_{pq}) = \frac{1}{1 - \frac{Avail\_Bandw(E_{pq}) + Network\_Bandw_i}{Link\_capacity(E_{pq})}}$$

where $Network\_Bandw_i$ is the projected network bandwidth required for object i. Therefore heavily used links are unlikely to be chosen in the new channels.

Given this information, the Resource Manager can then execute the actual placement algorithm. The algorithm creates an initial solution by placing the application objects sequentially. To do so it creates an absolute (but random) ranking for all object types.

- The Greedy algorithm:
  1. For each object i, perform the following steps:
     (a) Determine where to put the object in the domain.

i. To determine the location, the algorithm uses a modified network graph G'. To create this graph we remove from the network graph G all the links whose available capacity is below the bandwidth requirements of the candidate object. Note here, that we consider both the incoming and outgoing invocations of object i.
ii. Using G' find, for each possible location for object i, the sum the weights of the shortest paths to all these objects that invoke methods on object i, plus the paths to those objects whose methods object i will invoke.
iii. Choose the location that minimizes the sum of lengths of communication channels for the new object i and has not been considered yet in this iteration to host object i.
iv. Calculate the contribution to the processor's load and memory after hosting the new object. Consider this processor, only if the addition of the object i does not introduce an overloaded processor (load above `high`) and there is available memory for object i. Otherwise, mark this processor as unavailable and go back to step iii to select another processor.
v. Project new values for the processor load, processor memory and bandwidth on the communication links.

It can be shown that the greedy algorithm satisfies our desiderata. It is efficient, because we have to run the all-pairs shortest path problem l times. We can make the algorithm faster $(O(|E|l)$ if we accept an approximate solution for the optimal position of an object. The final solution respects the QoS guarantees for the application objects: if a solution is found, then all the communication links are expected to have enough available bandwidth for the object. It tries to optimize the network resources utilization by: trying to serve invoking objects that are closely located, optimizing the length of the communication links, and distributing the network traffic uniformly.

During operation, the Resource Manager may detect overloaded or insufficient resources to provide the quality of service required by the tasks. In such a case, the Resource Manager may choose to re-distribute some of the objects in the domain. Object migration may also be required to adjust in the new configuration of the system as it has changed over time or when a processor fails. The cost of object migration is justified with amortization over many method invocations, and constrains the rate at which objects are moved [6].

6 Distributed Real-Time Scheduling of Method Invocations

When application tasks invoke methods on objects across multiple processors and multiple domains, the degree to which end-to-end timeliness is achieved is a combination of two factors; the relative importance of the tasks and the objects invoked by the tasks and the collective timeliness as tasks execute methods over multiple processors. Timing requirements and importance metrics are not normally correlated. Consequently, scheduling tasks to maximize the probability that the most important tasks meet their deadlines is not a trivial problem.

Our scheduling algorithm is based on the least-laxity scheduling algorithm that has been proven [4] to be very efficient in multi-processor environments. In least-laxity scheduling the laxity of a task represents a measure of urgency for the task and is computed as the difference between the deadline and the projected computation time for the task. To increase the probability that important tasks meet their deadlines, our scheduling algorithm relaxes the timing constraint (laxity value) and considers the object importance when scheduling the methods invoked by the tasks on the processors. The order in which the methods are scheduled is driven by both the laxity value of the invoking tasks and the importance of the objects.

6.1 The Least Laxity Scheduling Algorithm

When a new task t arrives, the Resource Manager 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 task's initial laxity is computed based on information collected during previous executions of the task. If no such information is available, the application programmer must provide an estimated computation time required for the task to run. This information is stored in the task graph, kept by the Resource Manager, along with other information about the application 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 $Processing\_time_m$ for executing method m measured by the Profilers and the $Mean\_processing\_time_m$ calculated by the Resource Manager based on previous executions of method m. The optimality of least laxity scheduling compared to earliest deadline first scheduling, is due to the fact that least laxity scheduling considers the computation times of the tasks to derive the laxity values.

When an activity traverses a processor boundary, it carries with it the caller's scheduling parameters (laxity value) included in the message header. As the execution of the application task moves from processor to processor, its scheduling needs are carried along and honored by the scheduler on each processor. When the activity 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 activity's laxity value.

The adjustment of the laxity, $Laxity_t$, provides the feedback loop shown in Figure 3. This is the first level of the three-level feedback loop structure of the 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 millisecond by 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. The rest of the information flow, shown by light arrows in the Figure, form the second level of the feedback loop and operate more slowly, on a second-by-second basis. When a task completes, the task's remaining laxity is recorded as the residual laxity for the task.

6.2 Application Task States

Figure 4 shows the states of a task during its lifetime. These states result from the scheduling decisions made on the processors that host the objects whose methods are invoked by the task. Each task can be found in one of the following states:

- **Running**: A task is active when the method invoked by the task is currently executing. The Local Scheduler decides in favor of a new task making a local method invocation on the processor, only if the new task is more urgent (smaller laxity value) than the currently

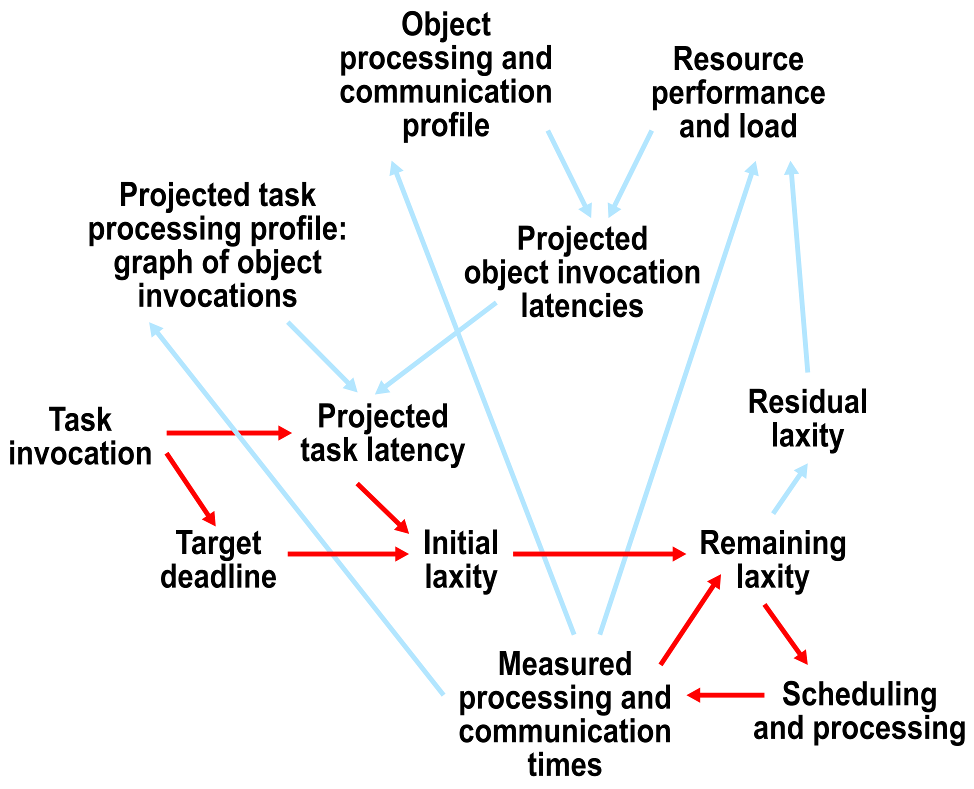

Object processing and communication profile
Resource performance and load
Projected task processing profile: graph of object invocations
Projected object invocation latencies
Task invocation
Projected task latency
Residual laxity
Target deadline
Initial laxity
Remaining laxity
Measured processing and communication times
Scheduling and processing

Figure 3: Scheduling and Profiling Loops of the Multiple Feedback Loop Structure.

executing task and the new task's laxity value is smaller than the remaining computation time of the currently executing task. In such a case, the current task will be suspended (task enters the `Waiting` state) and the new task will be dispatched for execution. When the currently executing method makes a remote method invocation on an object located on the same or a different processor, the task's execution propagates to the invoked object and the Local Scheduler orders the currently executing method on its queue according to its laxity value. When the task execution completes, the task becomes `Inactive`.

- **Waiting**: A task enters the Waiting state when the method invoked by the task is preempted by a method with a higher priority executing on the same processor. If, during execution, the task's laxity value becomes negative, this indicates that the task will miss its deadline even if the deadline has not passed yet. Our soft real-time system allows the task to continue execution even after its deadline, as long as it does not interfere with the execution of the other objects in the processor. The Local Scheduler schedules those tasks with negative laxity value during the time that the processor would otherwise have been idle.

- **Inactive**: A task remains in the Inactive state until the client thread starts executing again. Each time the task executes, the Resource Manager uses the ratio of the residual laxity to the initial laxity of the task during the task's previous executions, to estimate a new initial laxity value for the task.

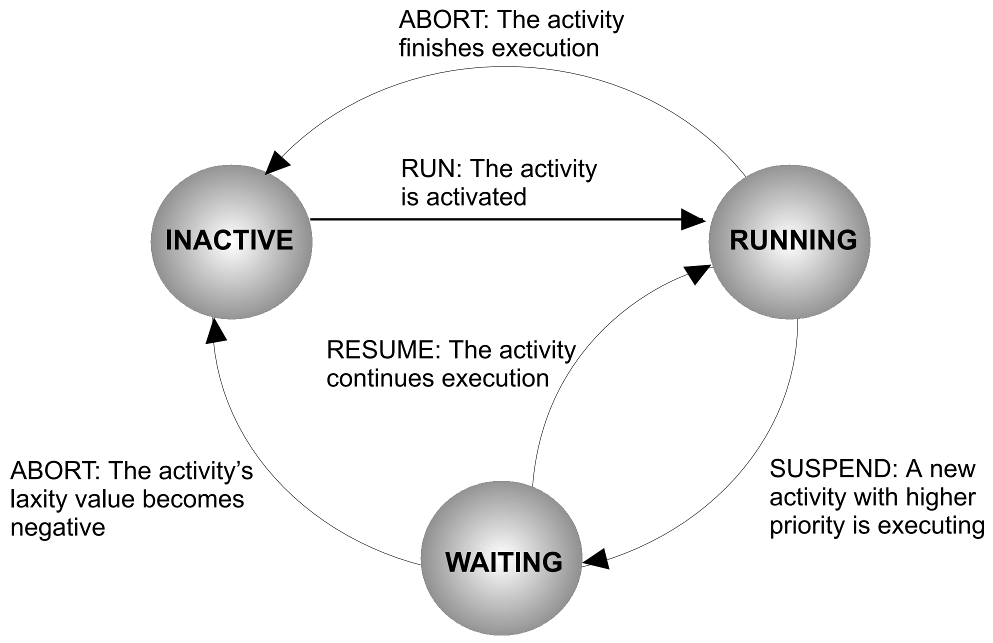

ABORT: The activity finishes execution
RUN: The activity is activated
INACTIVE
RUNNING
RESUME: The activity continues execution
ABORT: The activity's laxity value becomes negative
SUSPEND: A new activity with higher priority is executing
WAITING

Figure 4: The States of a Task During Its Execution.

7 Determining the Number of the Resource Manager Peers

When an application task t completes, the Resource Manager compares the residual laxity with the initial laxity of the task. If the Resource Manager's estimate for the projected latency of the task was accurate, the residual laxity will be the same with the initial laxity of the task. The Resource Manager uses the ratio

$$\frac{Residual\_Laxity_t}{Initial\_Laxity_t}$$

to adjust the estimates of the projected latency of the task. The Resource Manager can identify an overloaded domain as the domain in which the residual laxity of the tasks is frequently smaller than the initial laxity. In such a case, a Resource Manager peer is able to define a new resource management domain and create a new Resource Manager peer in the domain to offload some of its work. Although the new Resource Management domain was created by an existing Resource Manager, yet, it is autonomous and behaves like the rest of the domains.

To avoid uneccessary domain creations, the Resource Manager estimates the remaining time for some of its application tasks to complete. If the remaining time is small, the creation of a new domain will not increase the performance gain since it will be a new domain with a small number of application tasks and objects that will soon need to be merged. So, the creation overhead will outweight the benefit of balancing the remaining load.

The Resource Manager is an essential component of the domain and, therefore, must be replicated for fault-tolerance and high availability. Different replication styles can be used based on the level of reliability to be provided by the Resource Manager to the domain. Among the replication styles, active replication is usually prefered when faults are critical, when recovery must be fast, and when the cost of transferring the state is larger than the cost of executing the operation. In active replication, all the Resource Manager replicas will perform the same operations in the same order. To maintain consistency among the replicas, we use a group communication protocol to transparently translate method invocations on an object group into invocations on the individual object replicas **??**. Typically, the Resource Manager has between two and four replicas. The replication is transparent to the outside of the domain world.

To determine the number of Resource Manager peers, we are using a *ReplicationDegree* algorithm (Figure 5) that incrementally determines if a new Resource Manager peer needs to be created, based on the benefit that the addition on the peer has for the system. The need for more Resource Managers increases based on the number of objects requiring service in a domain and the QoS requirements during the transmissions.

7.1 Resource Manager Utility

The utility of a Resource Manager represents the significance of having that such a Resource Manager peer in the system. The utility value is computed based on the significance that the Resource Manager peer has for the system and the probability of faults of the Resource Manager peers.

Let $max\_dist_c = max_{i,j \text{ in the same domain}} Dist(object_i, object_j)$ be the maximum node distance between any two objects that belong to the same domain. We define the importance Imp_c of the creation of a Resource Manager peer c to be the sum of the number of the objects N serviced by the Resource Manager, weighted by the level of service $level_i$ the objects require and the maximum distance between any two objects in the domain:

$$Imp_c = \sum_{i \in c}^{N} Consumer_i * level_i * max\_dist_c$$

where $i \in c$ denotes that object i is serviced by Resource Manager c. Essentially, the higher the level of service provided to the more number of objects with the maximum node distance, the more important the Resource Manager becomes for a domain.

Assuming that r_c represents the number of Resource Manager peers and q_c is the failure probability for the Resource Manager within the domain, we compute the utility $Util_{r_c}$ of having an r_c^{th} Resource Manager peer c to be the product of the Resource Manager's importance and the probability of failure of all but one of the peers:

$$Util_{r_c} = Imp_c * \binom{r_c}{r_c - 1} * q_c^{r_c - 1} * (1 - q_c)$$

7.2 Replication Degree Algorithm

Thus, the problem is narrowed to determining the appropriate number for Resource Manager peers by computing the utility of having another Resource Manager peer to the system. At least one Resource Manager is required for the system. Our algorithm incrementally allocates a new Resource Manager peer, if the addition of the new Resource Manager will benefit the objects requiring service from this domain. Consequently, each time we allocate a new Resource Manager, we create new Resource management domain. Our algorithm is novel because the Resource Manager peers are independent, autonomous and co-operate in a purely decentralized world, there is no centralized component to manage all the application tasks in the system.

```
Replication Degree algorithm:
    calculate utility $Util_{r_c}$ of having $r_c^{th}$ replica of Resource Manager $c$
    allocate one Resource Manager $c$ for the whole system
    while maximum remaining UtilityValue $\geq$ lowUtilityBound
        find Resource Manager $c$ with maximum utility $Util_{r_c}$
        allocate an $r_c^{th}$ replica for Resource Manager $c$
        increment the number of replicas for Resource Manager $c$
return the number $r_c$ of replicas for the Resource Manager $c$
```

Figure 5: Pseudocode for the Replication Degree algorithm.

Since objects can be invoked by multiple tasks, there may be cases when the objects are invoked by tasks that originate from different domains. In such cases, we try to configure the resource management domains so that we include both of the tasks in the same domain.

To avoid the continuous creation of Resource Manager peers, we determine a lower bound *lowUtilityBound* to be the minimum utility value accepted for a peer. Resource Manager peers with utility values below this bound are not considered for allocation in the system. Essentially, the first few Resource Manager peers are the most important to the system. Essentially, we avoid having Resource Managers servising only a small number of objects. If during operation, some tasks finish execution, the Resource Manager with a small number of tasks will try to merge domains with other peers.

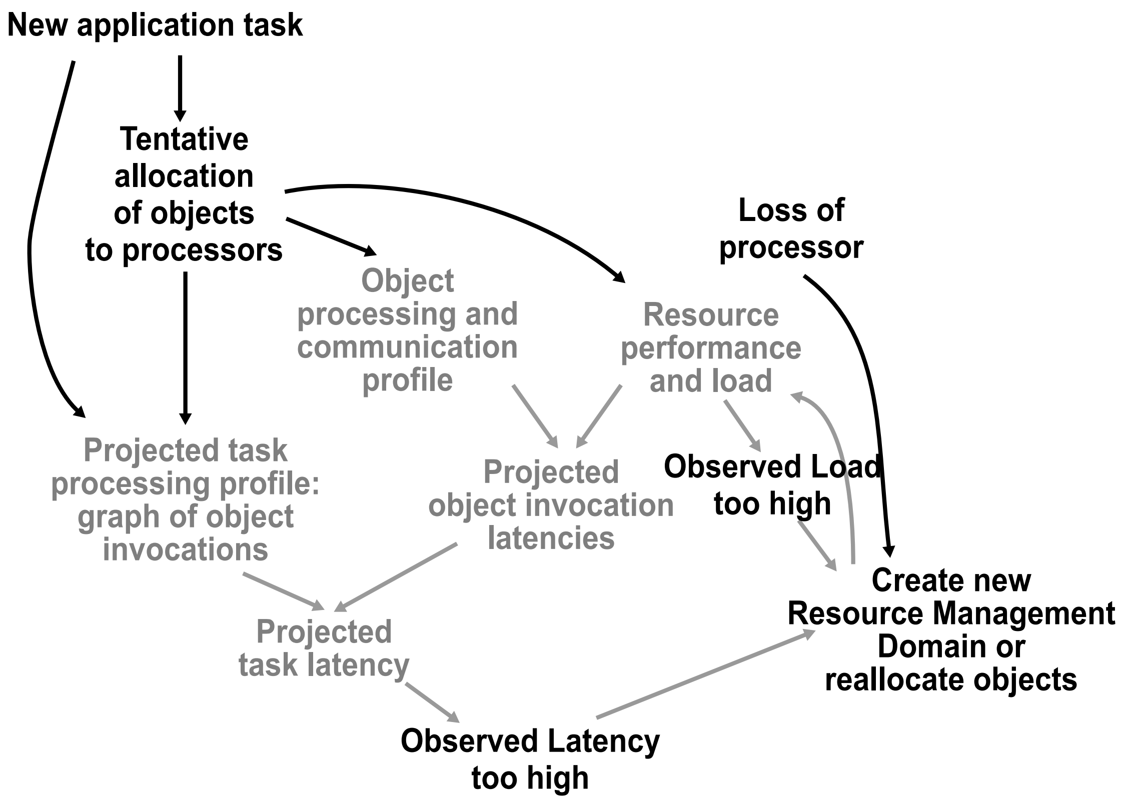
The location transparency CORBA provides, is a key element for our system. If a Resource Manager peer fails, the objects serviced by that Resource Manager, are automatically serviced by some other Resource Manager peer, which will likely be in a different network location and a different domain. Although this will introduce some delay in the transmission of the data, it will not interrupt the actual execution of the applications.

8 Multiple Feedback Loops

The Resource Management System uses a three-level feedback loop structure:

- The first level (milliseconds) uses measurements of elapsed time to refine the estimated residual laxity of executing tasks, which are used by the least laxity scheduler.
- The second level (seconds) uses measurements of elapsed time and measurements of processor loads to refine the initial estimates of the laxity for the tasks as they start.
- The third level (several seconds) uses measured processor loads, queueing and network latencies and measured residual laxities to revise the allocation of objects to processors and to create (or delete) Resource Manager peers for the system.

The distributed Resource Management system (Figure 6) operates on an allocation scheme that is adaptive. As the new application task runs in a domain (or across multiple domains), a profile of the method invocations for that task is constructed and is added to the Resource Manager(s)'s information base. The Resource Manager makes a tentative allocation of new objects to processors 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 processor and requires new projections of the task latencies. If the allocation is acceptable, it is actually performed. As the number of application tasks increase or the latency on a processor is too long,



New application task
Tentative allocation of objects to processors
Object processing and communication profile
Resource performance and load
Loss of processor
Projected task processing profile: graph of object invocations
Projected object invocation latencies
Observed Load too high
Create new Resource Management Domain or reallocate objects
Projected task latency
Observed Latency too high

Figure 6: Multiple Feedback Loops:The Creation of a New Resource Management Domain.

a Resource Manager peer creates a new peer to offload some of its work. If application tasks finish execution and objects are deleted from a domain, existing Resource Manager peers can be merged together.

9 Related Work

Many researchers have realized the need for systems that can adapt to dynamic, unpredictable changes in the computing environment. Nett *et al* [10] 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* [16] 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* [5] 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* [15] 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* [9] 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.

There is a large body of work [1, 2, 3, 12] that focuses on maximizing the bandwidth efficiency in delivering multimedia services. Zhao *et al* [17] exploit temporal and spatial structures to accomplish optimal stream multiplexing. Recent work on multi-cast routing algorithms is also related to our work [14]. However the problem we are solving is different. We are trying to find an optimal allocation for the objects on the processors, as the objects are invoked by multiple tasks and the tasks arrive dynamically.

10 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. The Resource Management System is based on a three-level feedback loop allows activities with different levels of temporal granularity, scheduling at the level of milliseconds, profiling over seconds, and object re-configuration over many seconds.

References

[1] M. Alfano and R. Sigle, "Controlling QoS in a collaborative multimedia environment," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY (Aug. 1996), pp. 340-347.

[2] S. Chatterjee, "A quality of service based allocation and routing algorithm for distributed, heterogeneous real time systems," *Proceedings of the 17th International Conference on Distributed Computing System*, Baltimore, MD (May 1997), pp. 235-242.

[3] P. K. Chrysanthis and D. Mosse, "Management and Delivery of Multimedia Traffic," *2nd Int'l Workshop on Community Networking Integrated Multimedia Services to the Home*, (June 1995).

[4] 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.

[5] 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.

[6] 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).

[7] 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.

[8] P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki and P. Narasimhan, “The Realize middleware for replication and resource management,” *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware '98*, The Lake District, England (September 1998), pp. 123-138.

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

[10] 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.

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

[12] S.V. Raghavan and S.K. Tripathi, *“Networked Multimedia Systems”*. Prentice Hall, Upper Saddle River, NJ, 1998.

[13] 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.

[14] H. F. Salama, D. S. Reeves, and Y. Viniotis, “Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks”, *IEEE Journal on Selected Areas in Communication*, vol.15, no.3, (April 1997), pp. 332-345.

[15] 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.

[16] 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.

[17] W. Zhao and S. K. Tripathi, “Bandwidth-efficient continuous media streaming through optimal multiplexing,” *Proceedings of the ACM SIGMETRICS '99, International Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA (May 1999), pp. 13-22.