To appear in the Proceedings of the First International Symposium on Agent Systems and Applications'99

CPU Resource Control for Mobile Programs*

Manoj Lal Raju Pandey
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis
Davis, CA 95616
{lal, pandey}@cs.ucdavis.edu

Abstract

There is considerable interest in developing runtime infrastructures for programs that can migrate from one host to another. Mobile programs are appealing because they support efficient utilization of network resources and extensibility of information servers. This paper presents a scheduling scheme for allocating resources to a mix of real-time and non real-time mobile programs. Within this framework, both mobile programs and hosts can specify constraints on how CPU should be allocated. On the basis of the constraints, the scheme constructs a scheduling graph on which it applies several scheduling algorithms. In case of conflicts between mobile program and host specified constraints, the schemes implements a policy that resolves the conflicts in favor of the host. The resulting scheduling scheme is adaptive, flexible, and enforces both program and host specified constraints.

1. Introduction

In computing models that support migration of programs, a program migrates to a remote host, executes and accesses the host's resources. For instance, Java [1] programs are increasingly being used to add dynamic content to a web page. When a user accesses the web page through a browser, the browser migrates the Java programs associated with the page and executes them at the user's site. Other examples of such computing models include the remote evaluation model [20] and the general purpose mobile programming model [7, 21]. The common element in these models is the ability of a runtime system to load externally defined user programs and execute them within the local name space.

In this paper, we focus on the problem of CPU resource allocation for mobile programs. The CPU allocation problem has been studied extensively and researchers have proposed several scheduling algorithms and scheduling schemes. A scheduling algorithm constructs schedules for applications that specify similar constraints on CPU resources. For instance, the Earliest Deadline First scheduling algorithm (EDF) [15] builds a schedule for real-time applications with deadline based constraints. A scheduling scheme, on the other hand, is a mechanism for combining different scheduling algorithms. Thus, a scheduling scheme constructs schedules for applications with different CPU constraints. For instance, the CPU inheritance scheduling scheme in [9] combines the real-time scheduling algorithm based on EDF with the multi-priority based round robin algorithm [23] for scheduling real-time and interactive applications.

Most scheduling techniques proposed to date attempt to allocate resources to applications on the basis of constraints, such as lower bounds and real-time deadlines, that the applications specify. Further, they try to satisfy overall objective functions such as fairness, responsiveness and CPU utilization. The resource allocation problem for mobile programs, however, has an additional component — host specified constraints on how resources should be used.

Two primary concerns drive host-specific resource usage constraints: *security* and *quality of service*. The security concerns of a host relate to preventing external programs from getting unlimited or unauthorized access to CPU resources, thereby staging denial of service attacks. The quality of service concerns involve providing differentiated levels of services to different categories of mobile programs.

*This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

For instance, a host may want to allocate more resources to mobile programs originating from its partner sites.

In this paper, we present a CPU scheduling scheme that controls the allocation of CPU resources to a mix of real-time and non real-time mobile programs by enforcing both mobile program and host specified constraints on how CPU should be allocated. Specifically, the scheme includes the following:

- *Specification of resource usage constraints*: The scheme includes a runtime interface and a specification language that mobile programs and hosts can use to specify resource usage constraints such as shares, priority, upper bounds and real-time deadline constraints.

- *Hierarchical grouping*: The specification mechanism can organize mobile programs into groups and subgroups on the basis of network domains, resources, or other host-specified groupings. This results in a hierarchical organization of mobile programs.

- *Scheduling algorithms*: The scheme includes three scheduling algorithms: (1) an algorithm for enforcing shares and priority constraints on non real-time mobile programs, (2) an algorithm for enforcing deadline constraints on real-time mobile programs, and (3) an algorithm for enforcing security constraints.

- *Algorithm composition policy*: The scheme uses an algorithm composition policy to resolve any conflicts among resource usage constraints. The policy aims at meeting a host's security needs and preferences first, followed by the other requirements of mobile programs.

The scheduling scheme is general: it can be integrated within HTTP servers, operating systems, and mobile programming runtime systems such as the Java Virtual Machine (JVM) [16, 14].

We have implemented the scheduling scheme within a simulation environment and the JVM (JDK version 1.1). Experimental results demonstrate that the scheme helps hosts to both protect and allocate CPU resources according to their preferences. The scheme effectively combines the scheduling algorithms in order to enforce both host and mobile program specified constraints.

This paper is organized as follows: In Section 2, we analyze the problem of CPU resource allocation for mobile programs in detail. Section 3 presents the scheduling scheme along with the different algorithms. In Section 4, we analyze the performance behavior of the scheme. We discuss related work in Section 5. Finally, we summarize our results and discuss future work in Section 6.

2. Resource allocation

The primary goal of a scheduling scheme is to construct a schedule that satisfies a set of resource usage constraints. A resource usage constraint specifies how resources should be allocated. We classify resource usage constraints both according to how mobile programs want to consume resources, and how a host wants to facilitate as well as protect usage of these resources. This results in two different views of resource allocation – client (mobile program) view and server (host system) view. We consider both views of resource allocation and the problems that arise when we try to satisfy both.

Notation: Let the term $\sigma_t(P)$ denote the amount of CPU allocated to a program P in a *schedulable unit*[1] t.

2.1. Client view

From a client program's perspective, requests for resources are driven by how the program demands and uses resources. Programs want to use as much CPU as possible so that they can perform their job quickly. They specify such requirements in terms of several constraints such as the following:

- **Lower bound**: A lower bound constraint, l, associated with a program P specifies that in case of contention, P be allocated at least l% of CPU in each schedulable unit.

$$\langle \forall\ t : \sigma_t(P) \geq (\frac{l}{100}) \times t \rangle$$

- **Weight**: A weight constraint, w, associated with a program P specifies that P be allocated w% of CPU in each schedulable unit.

$$\langle \forall\ t : \sigma_t(P) = (\frac{w}{100}) \times t \rangle$$

- **Share**: Shares are closely related to weights in that relative amounts of shares owned by a program define the program's weight constraint. Thus, if program P has s shares and the total number of shares in the system is S, then

$$\langle \forall\ t : \sigma_t(P) = (\frac{(\frac{s}{S})}{100}) \times t \rangle$$

- **Deadline**: Real-time constraints in the form of $<S,E,T>$ for a program P specify that between times S and E, P must get T amount of CPU.

$$\langle \sum_{t \geq S}^{t \leq E} \sigma_t(P) = T \rangle$$

[1] We divide the time line into schedulable units and specify resource usage constraints in terms of these schedulable units.

2.2. Server view

From the server's perspective, two concerns govern the allocation of resources – meeting client's needs and tightly controlling allocation of resources. Two factors unique to mobile environments accentuate the latter concern. First, a level of distrust exists between a host and mobile programs since mobile programs and hosts typically belong to different administrative domains. Mobile programs can maliciously disrupt a host by using unauthorized resources, by over-using resources, and by denying resources to other programs. Second, in distributed systems such as the web [3], hosts may provide varying degree of services to clients. A host may differentiate requests from different clients and allocate resources to these requests in accordance with the kinds of services the host wants to provide. For instance, a host may reserve 85% of its resources to mobile programs that originate from the paying customer sites and allocate the rest for other programs.

In mobile programming models [7, 21], the resource allocation problem has an additional dimension: A mobile program can circumvent resource control by migrating to another host and then returning to its previous host for more resources. Resource usage constraints, thus, apply not only to specific executions but to all executions of a program. We refer to such constraints as *lifetime constraints*. An issue closely related to lifetime constraints is that of uniquely identifying and authenticating a mobile program when it remigrates to the host system. The paper does not address this issue though we recognize that without sound authentication, a mobile program can easily change its identity and carry out denial of service attacks.

Thus, a host must differentiate and categorize mobile programs if it intends to impose resource constraints on them. In addition to the lower bound, shares and weight constraints, a host may also specify the following constraints:

- **Priority**: A priority constraint specifies that, given a set of programs to schedule, a host selects the program with the highest priority.

- **Upper bound**: An upper bound constraint, u, associated with a program P specifies that, within each schedulable unit, P be allocated at most u% of CPU.

$$\langle \forall\ t : \sigma_t(P) \leq (u/100) \times t \rangle$$

  A host can also specify upper bounds in absolute form.

- **Life-time constraints**: A lifetime constraint, l associated with a program P specifies that P be allocated at most l units of CPU time over all executions of P.

2.3. The scheduling problem

The CPU resource control problem is, therefore, one of developing a scheduling scheme that, given client and server resource usage constraints, constructs a schedule that enforces the constraints.

2.3.1. Specification of constraints. We have developed a runtime interface and a specification language that clients and servers use to specify resource usage constraints. We briefly describe them below.

We organize mobile programs into groups and subgroups. For instance, a group `ucdavis.edu` denotes all mobile programs that originate from this domain. This group may contain subdomains such as `cs.ucdavis.edu` and `ece.ucdavis.edu`. The notion of groups and subgroups results in a hierarchical partitioning of mobile programs. Clients can define their own groups and determine constraints for individual jobs within the group. Hosts can specify resource constraints on groups, subgroups, or individual mobile programs.

Mobile programs and hosts can specify the following sets of constraints:

- **Client constraints:** Real-time constraints and shares.

- **Host constraints:** Shares, priority, absolute upper bound, and lifetime constraints.

- **Dynamic constraints:** Both mobile programs and hosts can change their constraints dynamically. The constraints can be specified as functions of the state of the system. Currently, we consider constraints that can vary on only two state variables: the number of mobile programs belonging to a group and the time of day.

We omit the description of the constraint specification language here due to lack of space.

3. Resource allocation scheme

We first describe the overall approach for scheduling resources to mobile programs.

- Construction of scheduling graph: The scheme partitions mobile programs into real-time and non real-time programs. It captures the group-subgroup relationships along with the various constraints to construct a scheduling graph.

- Application of algorithms: The scheme applies three algorithms to the scheduling graph: (i) an algorithm to enforce upper bound and lifetime constraints; (ii) an algorithm to enforce share and priority constraints; and (iii) an algorithm to enforce real-time deadline based constraints.

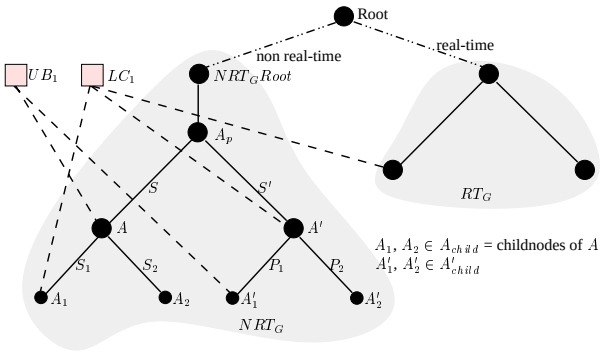


Figure 1. The scheduling graph

- Monitoring of system state: Since the host can specify constraints as a function of state variables, the scheduling scheme monitors the state of the system and adapts to the changes in the resource constraints by modifying the scheduling graph.

In the following sections, we describe the individual algorithms and how they are composed to build the scheduling scheme.

3.1. Construction of scheduling graph

The scheduling scheme first builds a *scheduling graph* (Figure 1) from resource usage constraints. The scheduling graph is made of three subgraphs: (i) Real-time, (ii) Non real-time, and (iii) upper-bound. The real-time subgraph is a single node (a real-time guarantee group, RT_G) containing all programs that specify deadline based constraints.

The non real-time subgraph, NRT_G in Figure 1, is a hierarchical graph, where each node denotes a group and each edge the group-subgroup relationship. Mobile programs are at the leaves of NRT_G. The edges of NRT_G are annotated with share or priority constraints. For instance, in Figure 1, the label on edge (A_p, A) specifies that group A has S shares. Similarly, the label on (A', A'_1) specifies that mobile program A'_1 has priority P_1. The amount of shares or priorities allocated to a group node is relative to its parent group. For instance, group A_1's share S_1 of CPU resources are with respect to the CPU resources allocated to its parent group, A. This results in a modular allocation of CPU: Any changes in the share allocations to A_1 or A_2 do not affect the CPU allocations to programs that belong to different groups, for instance A'.

The upper bound subgraph represents the security constraints. Nodes in this subgraph denote specific upper bound and lifetime constraints. Edges link these constraints to the relevant groups and mobile programs. Upper bound and lifetime constraints are general in that they can encapsulate more than one node in the scheduling graph. Moreover, the nodes encapsulated by a particular constraint need not be at the same level. For instance, as shown in Figure 1 the upper bound constraint UB_1 applies to group A and mobile program A'_1. There is a need for such constraints so that the host can control mobile programs belonging to different levels in the hierarchy. For example, suppose a site wants to impose an upper bound constraint that mobile programs accessing a particular database be allocated at most 10% of CPU. Such mobile programs may exist in different groups and may span multiple subtree domains.

Hosts define the backbone of the scheduling graph – a graph consisting of empty RT_G and possibly non-empty NRT_G. When a new client program arrives, the host can create a new group node (A_p) for the client, specify constraints for A_p, and add A_p at an appropriate place in the scheduling graph. The client program can create subgroups under A_p and define resource usage constraints for the subgraph under A_p. Note that the scheduling graph is dynamic; it changes whenever mobile programs arrive and when client-specific and host-specific resource usage constraints change.

3.2. Application of algorithms

The scheduling scheme operates on the scheduling graph to allocate CPU to mobile programs. An important aspect of a scheduling scheme is how conflicts between the mobile program and host specific resource usage constraints are resolved. A scheduling scheme must include a set of policies, called *algorithm composition policies*, that resolves any conflicts. Our scheduling scheme implements a policy that always resolves conflicts in favor of host constraints. The policy is summarized as follows:

- The scheme first ensures that constraints related with the security aspects of the host are satisfied. Thus, it always applies the upper bound algorithm first to enforce the upper bound and lifetime constraints even if it means that other programs do not get their requested CPU allocation or that some deadlines are missed.
- Next, the scheme enforces host-specified priority and share constraints in order to implement host's preferences.
- Finally, the scheme schedules non real-time jobs according to their relative shares, whereas it schedules real-time jobs so that their deadlines constraints are met.

The scheme partitions the continuous time line into small *quantum time chunks* (Figure 2). Within each quantum time chunk, it schedules mobile programs from RT_G according to their reservations. The reservations fix the times when the

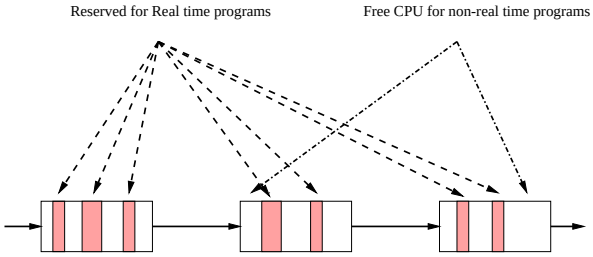


Figure 2. List of quantum time chunks with reservations for real-time programs

scheme allocates CPU to real-time programs. This is shown as shaded parts in a single quantum time chunk. The scheme then allocates the remaining time to non real-time programs. The scheduling of non real-time mobile programs starts from the root node of NRT_G graph (NRT_GRoot). The scheme traverses from NRT_GRoot to one of the leaves of the graph.

Figure 3 describes the overall working of the schedule function for one single quantum time chunk. In the next sections we describe the individual algorithms for non real-time and real time programs.

3.2.1. Scheduling of non real-time programs. The crux of the algorithm for non real-time programs is the decision associated with the children nodes of a node. If the children nodes have priority based constraints, the algorithm selects the child node with the highest priority. If the children nodes have share based constraints, the algorithm selects a child node on the basis of the share allocations of the children nodes.

The algorithm to allocate CPU on the basis of share based constraints extends the ideas in the SMART scheduling algorithm [17] to a hierarchy. SMART defines two numbers for each application – a *virtual time (VT)* and a *virtual finish time (VFT)*. The notion of VT and VFT was originally developed in fair queuing algorithms for congestion control in network protocols [13] and has been used in CPU scheduling in SMART and Stride scheduling [25].

We extend the notion of virtual time to define three entities – an *upper virtual time (UVT)*, a *virtual finish time (VFT)*, and a *lower virtual time (LVT)*. First, we present the intuition behind virtual time and virtual finish time and give their formal definitions as they are used in the SMART system. We then define UVT, VFT and LVT.

In the SMART system applications are partitioned into different priority queues. Applications within each priority queue have shares. The system associates a VT with each application and priority queue.

- *VT of priority queue P:* Initially:

$$VT_P(t) = 0 \tag{1}$$

  At a later time, if an application within P was initiated for execution at time (τ) and is currently (t) executing:

$$VT_P(t) = VT_P(\tau) + \frac{t - \tau}{\sum_{A \in P_{member}} S_A} \tag{2}$$

  where A is an application in P and S_A represent A's shares.

- VT of application: When an application A joins the priority queue P for the first time at time t:

$$VT_A(t) = VT_P(t) \tag{3}$$

  At a later time, if A was initiated for execution at time (τ) and is currently (t) executing:

$$VT_A(t) = VT_A(\tau) + \frac{t - \tau}{S_A} \tag{4}$$

  where S_A represent A's shares.

The virtual time of an application measures the degree to which the application has received its proportional share of CPU on the basis of its share allocation. The difference between $VT_A(t)$ and $VT_P(t)$ gives a measure of whether A has received its share-based allocation. If $VT_A(t)$ is less then $VT_P(t)$, A has received less than its share and vice-versa. The virtual time for an application advances at a rate inversely proportional to the number of shares it holds. If an application has a large number of shares, its virtual time will increase at a smaller rate and, therefore, the application will be scheduled more often to make its virtual time same as that of the queue.

The virtual finish time of an application refers to its virtual time if the application had been selected for the currently executing time quantum.

- VFT of application: When application A joins queue P at time τ:

$$VFT_A(\tau) = VT_P(\tau) + \frac{Q}{S_A} \tag{5}$$

  where Q is the time quantum. Later, when A has been scheduled for some time and now is going to be stopped (t):

$$VFT_A(t) = VFT_A(\tau) + \frac{Q}{S_A} \tag{6}$$

  where τ is the time when VFT_A was last changed.

```
for (;;) {
    t = next_time_quantum();
    while (in current quantum time chunk) {
        if (time to schedule a real-time mobile program) {
            Check for upper bound and lifetime constraints;
            schedule the mobile program;
            update lifetime and upper bound constraints;
        }
        else {
            //schedule from the remaining (non real-time) hierarchy
            currentnode=Root of the non real-time programs;
            while (currentnode is not a leaf node) {
                Check for upper bound and lifetime constraints;
                if (constraints of childnodes based on priority) {
                    currentnode = select childnode with earliest priority;
                }
                if (constraints of childnodes based on shares) {
                    currentnode = select childnode on the basis of shares;
                }
            }
            schedule the leaf node;
            update lifetime and upper bound constraints;
        }
    }
}
```

Figure 3. The scheduling scheme

A property of virtual finish time is that it does not change while the application is executing. It changes only when the task is rescheduled. The scheduling algorithm selects the application with the smallest virtual finish time from the highest priority queue for scheduling.

To extend the idea of virtual time to a hierarchy, we define three quantities: upper virtual time (UVT), virtual finish time (VFT) and lower virtual time (LVT) for each node in the hierarchy. The reason we require UVT and LVT is that in NRT_G, each internal node is both a child node and a parent node. UVT of the internal node is compared with the LVT of the parent node to select the child node that should be scheduled.

Assume that the algorithm has reached a particular internal node A and the children nodes of A have share based constraints associated with them (Figure 1). Let the parent of A be A_p, and let A own S shares under A_p. Let A_{child} be the set of children nodes of A. Also, let each A_i in the set A_{child} own shares S_i.

- UVT: When A joins the hierarchy for the first time at time t:

$$UVT_A(t) = LVT_{A_p}(t) \tag{7}$$

  Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing:

$$UVT_A(t) = UVT_A(\tau) + \frac{t-\tau}{S} \tag{8}$$

- LVT: The lower virtual time at A selects one of A's children. Initially, when A joins the hierarchy,

$$LVT_A(t) = 0 \tag{9}$$

  Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing,

$$LVT_A(t) = LVT_A(\tau) + \frac{t-\tau}{\sum_{a \in A_{child}} S_a} \tag{10}$$

The UVT of a node measures the degree to which the node has received its proportional share of CPU from the parent node. The difference between UVT of a node and LVT of the parent node gives a measure of whether the node has received its share-based allocation. If the node's UVT is less then the parent node's LVT, the node has received less than its share and vice-versa. UVT advances at a rate inversely proportional to the number of shares the node holds. If a node has a large number of shares, its UVT

will increase at a smaller rate, and therefore it will be selected more often to make its UVT same as LVT of the parent node.

The virtual finish time of a node refers to its UVT, had the node been selected for the current quantum for execution.

- VFT: The VFT of a node A is its UVT had A been selected for the current quantum. When A joins the hierarchy for the first time at time t:

$$VFT_A(t) = UVT_A(t) + \frac{Q}{S} \tag{11}$$

  where Q is the quantum size. Later, when a mobile program from within A was initiated for execution at time τ and now (t) some other program is going to be scheduled:

$$VFT_A(t) = VFT_A(\tau) + \frac{Q}{S} \tag{12}$$

A property of the virtual finish time is that it does not change while the application is executing. It changes only when the task is rescheduled.

The algorithm selects the child node with the earliest virtual finish time (VFT). To summarize: The scheduling of non real-time programs starts at the root of the non real-time programs (Figure 1). The algorithm starts at the root and traverses the tree till it reaches a leaf, which represents a mobile program. At an internal node A, the algorithm examines the constraints associated with the children nodes of A. If the children nodes have priority based constraints associated with them, the algorithm selects the child node with the highest priority. If children nodes have share based constraints, the algorithm selects the child node with the earliest VFT. If the node selected is a mobile program, it is scheduled for execution, otherwise the process is repeated.

3.2.2. Scheduling of real-time programs. Real-time mobile programs are members of RT_G. The scheduling of mobile programs in RT_G is based on the scheduling algorithm in Rialto [12]. Rialto uses a *precomputed scheduling graph* to implement continuously guaranteed CPU reservations with application defined periods, and to guarantee time constraints. Applications make *CPU reservations* in the form of "reserve X units of time out of every Y units". Real-time applications request CPU resources by specifying *time constraints* of the form <S,E,T>. On the basis of the CPU reservations, Rialto constructs a `Rialto scheduling graph`. The nodes in the Rialto scheduling graph indicate either reserved time periods for applications or free time not reserved for any application. The time constraints for threads are then satisfied from the reserved time periods and from any free time that might be available.

Our real-time scheduling problem differs from the problem solved in Rialto in the following ways: First, we use simpler real-time constraints. We don't consider continuous *CPU reservations* of the form "reserve X units ....". Instead, we define CPU reservations over discrete base periods, i.e., quantum time chunks. With this simplification, there is no need to compute the *Rialto scheduling graph*. However, the RT_G algorithm is now more general, as CPU reservations can be carried out from any place in the base period rather than from some fixed locations in the Rialto scheduling graph. Further, our real-time scheduling algorithm must satisfy additional constraints in the form of upper bounds.

Resource allocation for real-time programs is done on the basis of a set of constraints of the form:

- RT_G`.upperbound = val1`: An upper bound on the time reserved for RT_G within each quantum time chunk. This prevents starvation of non real-time programs.
- `group.`RT_G`-bandwidth = val2`: Groups can reserve bandwidth within RT_G so that deadline based constraints for member mobile programs can be satisfied from the reserved bandwidth.
- `mobileprogram.deadline =` $<S,E,T>$: A mobile program within a group can request that its time constraints be satisfied by utilizing the bandwidth reserved for its parent group. If there is no bandwidth reserved for the parent group, the program will get only unreserved RT_G bandwidth to satisfy its constraints.

The scheduling algorithm allocates time within the quantum time chunks to satisfy reservation requests. The use of quantum time chunks is similar to the notion of *slot lists*[18]. While the slot list method considers only real-time applications, in our case the amount of CPU time available in each quantum time chunk is constrained by the upper bound on RT_G.

The real-time algorithm first reserves the bandwidth for each group in each quantum time chunk. For each <S,E,T> constraint, the scheduling algorithm makes reservations in the quantum time chunks (Figure 2) that fall within times S and E. The algorithm reserves the computation time T from within the parent group's reserved bandwidth, if any, and any free unreserved RT_G bandwidth that might be available within the quantum chunk. It does so by creating reservation nodes in each quantum time chunk. The reservation nodes specify the start time, the time reserved, and the mobile program for which the time has been reserved.

When a new real-time program arrives, the algorithm performs a feasibility check to determine if the deadline request can be met. The algorithm goes through the list of quantum time chunks, reserving any available RT_G time for the request. If the program's deadline cannot be met, any

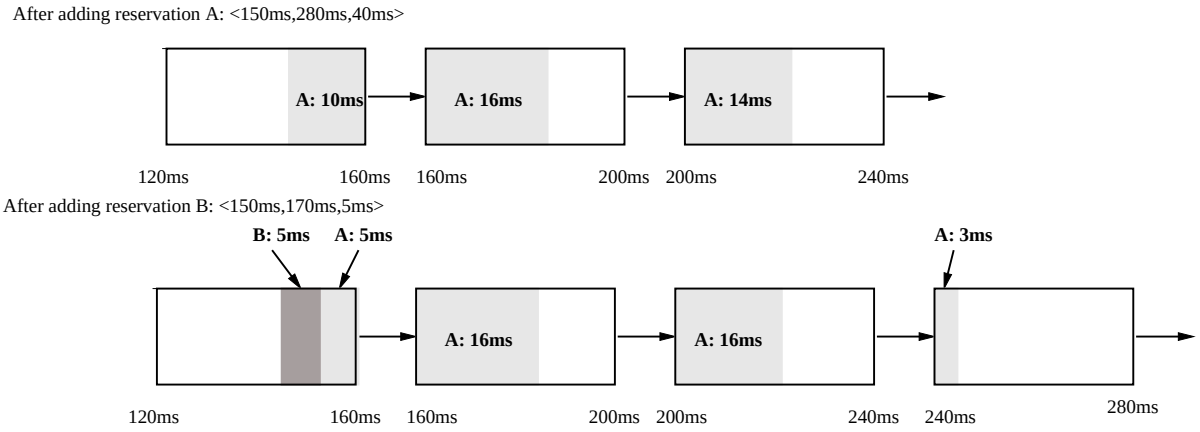


Figure 4. List of quantum time chunks for two reservations

reservation made for the program is freed. In the process of carrying out the feasibility checks, the algorithm performs a rearrangement of any reservations already made for earlier programs so that the deadline based constraints are added in the Earliest Deadline First (EDF) [15] order. Using EDF for adding new reservations increases the number of reservations satisfied.

In Figure 4, we show how the algorithm makes reservations for two requests: reservation A (<150,280,40>) and reservation B (<150,170,5>) in that order. We assume that the size of a quantum time chunk is $40ms$ and the host specifies an upper bound of 40% ($16ms$) for RT_G. Also, assume that the entire CPU time ($16ms$) for RT_G is available to the mobile programs. When request A is made, the algorithm greedily reserves any RT_G time available to A. When request B arrives, the algorithm rearranges the reservation for A so that the constraints of B can also be satisfied. This is done because B has an earlier deadline. If there were no rearrangement, B cannot be guaranteed since all RT_G time will already be used by A. Figure 5 describes the algorithm for making a real time reservation.

3.2.3. Resource usage control. The upper bound subgraph captures the upper bound and lifetime constraints on groups and mobile programs. Each security node in the graph maintains the usage information for the groups and programs that the node monitors. As the scheduling scheme traverses the scheduling graph, it checks the security node associated with a node before it applies any scheduling algorithm to the node. If selecting a program from within that node will cause an upper bound or a lifetime constraint to be violated, the particular internal node is not selected. For example, assume that the scheme decides to schedule a program in the subtree under A_p in Figure 1. Before it decides between nodes A and A', the scheme checks with the security nodes that control A and A' (UB_1 for A and LC_1 for A') to ensure that the two nodes do not violate any constraints. The scheme then employs the selection algorithm as described earlier to select one of the two.

4. Implementation and performance analysis

To assess the behavior of the scheduling scheme, we first implemented the scheme as part of a simulation engine and conducted several experiments using the simulation engine to analyze the performance behavior of the scheme. Once we were satisfied with the scheme, we then implemented the scheduling scheme within the Java virtual machine (JVM). We analyzed its behavior within the JVM as well. In this section, we first describe the two implementations. We then present the performance behavior of the scheme on the two implementations.

4.1. Implementation

The simulation engine: The simulation engine provides an API for creating groups, specifying group memberships, constraints and mapping the constraints to the groups. After reading the various specifications, the simulation engine builds a scheduling graph, creates virtual threads for mobile programs, and simulates the scheduling of the virtual threads. We simulate time by keeping a virtual timer. Whenever a virtual thread is selected for scheduling, we advance the virtual time by the scheduling quantum and charge the quantum to the virtual thread. We also keep an event list for time based events. The event list is consulted every time before scheduling a thread. The list stores events that may signify a change in some (time based) constraint or the scheduling of a real time program.

Java virtual machine: We have modified the Java vir-

```
int reserve(S,E,T) {
    //get the set of real time requests interfering with the current reservation,
    and that have later deadlines than the current request.
    I = set of interfering requests;
    Remove_reservations(I); // remove reservations for I
    //see if current request can be satisfied
    int result = Try_adding(S,E,T);
    if (result = true) {
        //if previous requests can still be satisfied
        int result = Try_adding(I);
        if(result = true)
            return result;
        else {
            //not able to satisfy previous requests
            //with the new one, revert to earlier situation
            Remove_reservations(S,E,T);
            Try_adding(I);
            return false;
        }
    }
    else {
        // not able to satisfy new request, revert to earlier situation
        Remove_reservations(S,E,T);
        Try_adding(I);
        return false;
    }
}

int Try_adding(S,E,T) { //add the new request <S,E,T>
    t = get_time_quantum(S);
    t'= get_time_quantum(E);
    while (all T not reserved) {
        Check for upper bound on RT_G in current quantum time chunk;
        if (upper bound on RT_G not reached)
            reserve any RT_G time available;
        if(all T not reserved) {
            t = next_time_quantum();
            if(t > t') //unable to satisfy the current request
                return false;
        }
    }
    return true;
}
```

Figure 5. Algorithm for making real time reservations

tual machine (Solaris JDK version 1.1) to incorporate our scheduling scheme. The modified JVM contains an API for specifying groups, subgroups, and various resource usage constraints. In addition, it includes a thread API for managing and scheduling threads. We have integrated the notion of groups in our scheduling scheme with that of *ThreadGroups* in the JVM. The current implementation does not include support for scheduling real-time programs since the JVM currently does not have support for real time programs.

We implement the scheme in the JVM by separating user threads, which represent mobile programs, from system threads, such as the garbage collector. Our implementation scheme fully manages the scheduling and execution of user threads through a scheduling graph. For instance, if the state of a user thread changes from `runnable` to `suspended` and vice-versa, we use our thread API to make the thread runnable or blocked.

4.2. Performance analysis

We conducted several experiments on the simulation engine and the JVM. The goals of these experiments were to examine the effectiveness of the scheme in (i) satisfying both real-time and non real-time constraints; (ii) enforcing upper bound and lifetime constraints; and (iii) satisfying constraints that change dynamically. We describe the experiments and the results below. In all the experiments, the time quantum for a program is $5ms$.

4.3. General scheduling behavior

The first experiment demonstrates how the scheme schedules groups of mobile programs that are constrained by shares, priorities and upper bounds. Further, it shows how the upper bound constraints interact with shares and priority constraints. We performed this experiment on the modified JVM. In Figure 6(a), we show the hierarchy constructed from the client and host resource usage constraints. In Figure 6(b), we show the relative CPU allocations of groups G_0, G_1 and G_2. We also show the relative CPU allocations of mobile programs MP_9, MP_{10}, and MP_{11}.

Between times $(0, 35)$, G_0, G_1 and G_2 get 30%, 60% and 10% of the CPU respectively which matches their share allocations. At time $35s$, G_0 reaches its upper bound. This results in relative allocation for G_1 and G_2 to increase to 86% and 14% respectively that corresponds to the share ratio of $60 : 10$. When the upper bound of G_2 is reached, G_1 is the only group and it gets all the CPU resources till its upper bound is achieved as well.

Within G_0, the relative allocations of mobile programs MP_9 and MP_{10} are 20% and 80% respectively, accordingly to their share allocations. MP_{11} is not scheduled in the beginning because it belongs to a lower priority group. At time $25s$, the upper bound for PG_3 is achieved and then mobile programs from PG_4 are scheduled till the upper bound for G_0 is reached.

The scheme, thus, effectively implements relative allocations of resources within hierarchies of groups. Further, it enforces upper bounds constraints as well. Note that changes in CPU allocation to MP_9, MP_{10} and MP_{11} (programs in G_0) does not affect the allocation to G_1 or G_2. This highlights the modularity of the scheme.

4.4. Adaptivity

In the second set of experiments (conducted on the JVM), we show that the scheme dynamically adapts to changes in resource constraints. We use the scheduling graph of Figure 6(a) for the experiments. The share and the priority constraints are as specified in the graph. We remove the upper bound constraints for these experiments. The first experiment, depicted in Figure 7(a), demonstrates the allocation of CPU to the programs when the resource usage constraints depend on the number of mobile programs. The share constraint for G_2 (S_2) is as follows: if number of mobile programs is less than 3, then G_2 is allocated 10 shares else it is allocated 110 shares. At time $40s$, a new mobile program is added to G_2, resulting in a change in relative allocation. At time $120s$, one of the programs of G_2 finishes execution, and the relative allocations go back to their initial values. Later, when all threads of G_1 finish execution, the relative allocations for the other groups increase.

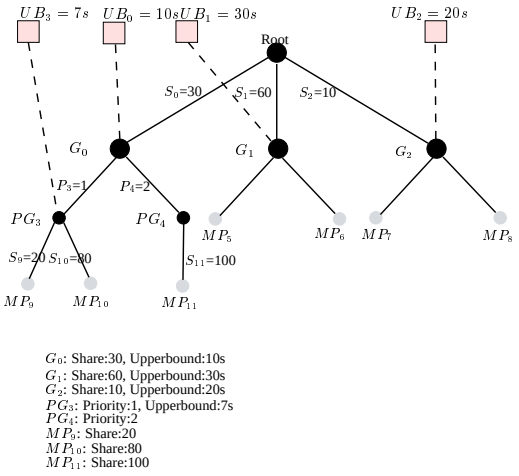
The second experiment (Figure 7(b)) demonstrates the allocation of CPU when constraints are time dependent. The following are the constraint values for the groups:

S_0: 30;
S_1: 50;
S_2: If $time \in [0s, 47s]$ then 20
else if $time \in [47s, \infty]$ then 110.

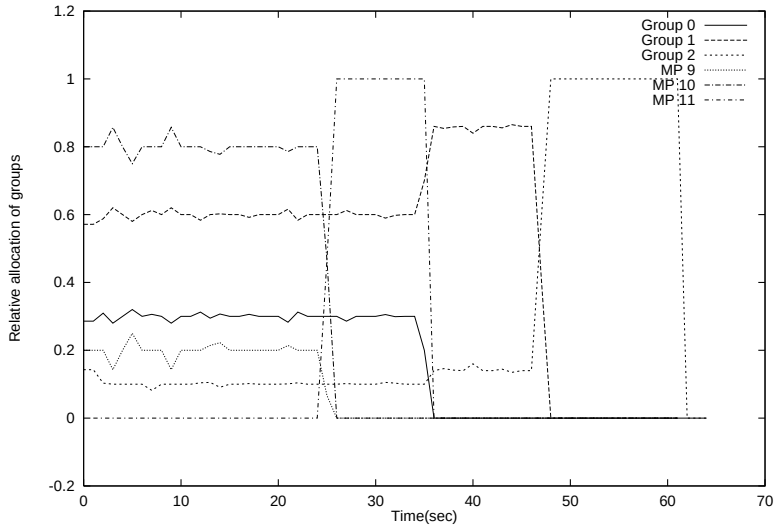
At time $47s$, relative allocation for groups change according to the specifications. At time $180s$, G_2 finishes execution and the relative allocations for other groups increase.

4.5. Lifetime constraints

This experiment (conducted using the simulation engine) demonstrates how the scheme enforces lifetime constraints. Figure 8 shows the scheduling graph. MP_4 has lifetime constraint of 2 migrations, and a total lifetime usage of $10s$. MP_4 also has upper bound of $3s$. At time $20s$, MP_4 leaves the host site. It migrates back at time $25s$. It again leaves at time $40s$ and migrates back at time $45s$. It finally leaves at time $55s$ and is not allowed to execute when it migrates

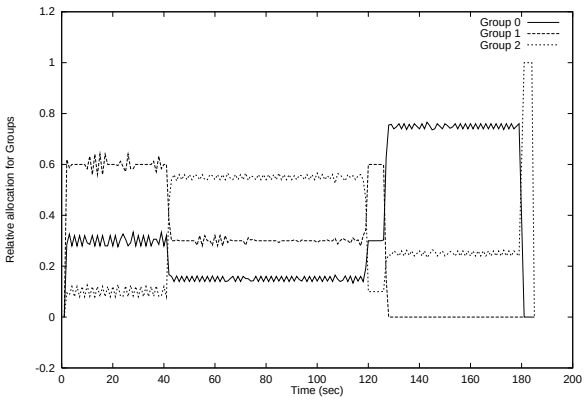


(a) The scheduling graph

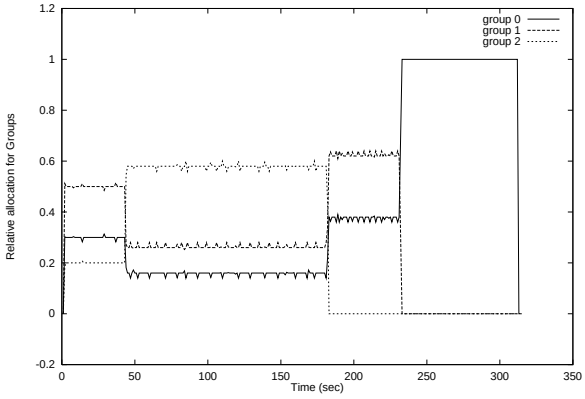


(b) Relative allocation of CPU for groups and mobile programs

Figure 6. General scheduling behavior of the scheme



(a) Constraints as a function of number of mobile programs



(b) Constraints as a function of time

Figure 7. Dynamic nature of the scheme

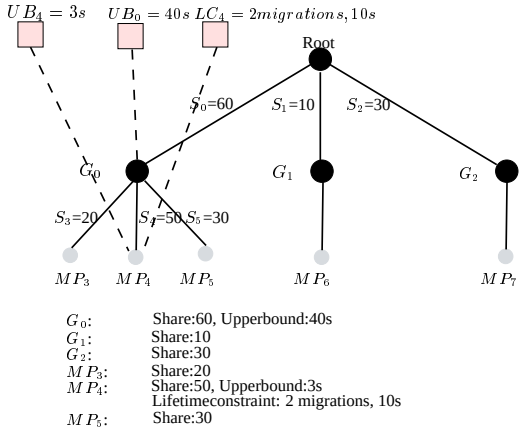


Figure 8. The scheduling graph for lifetime constraints

back for the third time. Figure 9(a) shows the relative allocations of mobile programs within $Group_0$. The relative allocations for MP_3 and MP_5 go up when MP_4 leaves. Allocation to any of the programs within the group stops when the group's upper bound is reached. Figure 9(b) shows the actual allocation for MP_4. The execution of MP_4 stops when its upper bound is reached. Then when it migrates back, it again gets allocated till its upper bound is reached. After the third time, the mobile program is not allocated anymore since its lifetime constraint of 2 migrations has been achieved.

4.6. Real-time programs

In the third set of experiments, we test the scheme's effectiveness in enforcing deadline based constraints for real-time programs. We conducted the experiments within the simulation engine.

For the first experiment (Figure 10(a)) we simulate the execution of an application that displays real time video streams from the local storage. The video input stream contains frames in JPEG compressed format at 15 frames/sec. We assume that the estimated execution time per frame to be about 30ms [17]. The application makes reservation requests for each frame within a 100ms period. If the reservation is granted then the application displays the frame; otherwise it skips the frame. The graph (Figure 10(a)) shows how the upper bounds and reserved bandwidth affect real time applications. The individual plots in the figure show the number of JPEG frames rendered per second as a function of the reserved bandwidth for the application. The different plots correspond to the upper bound set on the RT_G group in each quantum time chunk. As the amount of reserved bandwidth decreases, the number of frames rendered/second also decreases.

The second experiment (Figure 10(b)) demonstrates how the scheduling of real-time programs takes place in the presence of non-real time programs. There are two non-real time groups: $Group0$ and $Group1$ have shares 40 and 20, respectively. There are three programs with real-time reservations:

MP6: <1.1s,1.15s,10ms>
MP7: <1.12s,1.5s,70ms>
MP8: <1.15s,1.18s,5ms>

The host specifies an upper bound of 40% ($16ms$) on the RT_G group for each quantum time chunk of $40ms$. The plot shows that the real-time programs are allocated according to their reservations. At the same time non-real time programs are allocated according to their shares. Also, since there is an upper bound on RT_G group, real-time programs cannot starve the non-real time programs, even though real time programs are scheduled for more than $5ms$ (the default time quantum) at a given time.

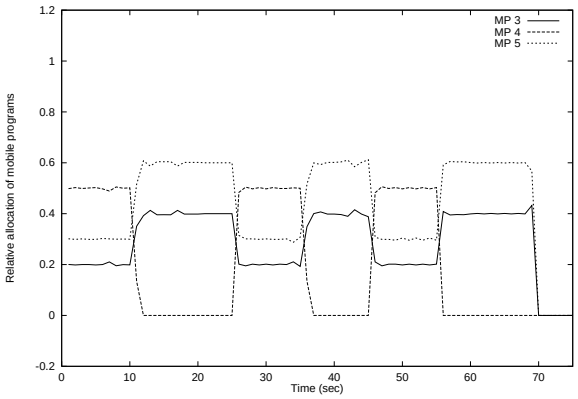
5. Related work

The subject of resource scheduling in general and CPU scheduling in particular has been widely studied. [6, 2] present a taxonomy of the different CPU scheduling algorithms. The scheduling techniques range from simple algorithms such as first come first served and priority queues [23] to more general, flexible and modular schemes [11, 9, 10, 24]. We compare our scheduling scheme and the algorithms with only those approaches which we believe are closest to our approach.

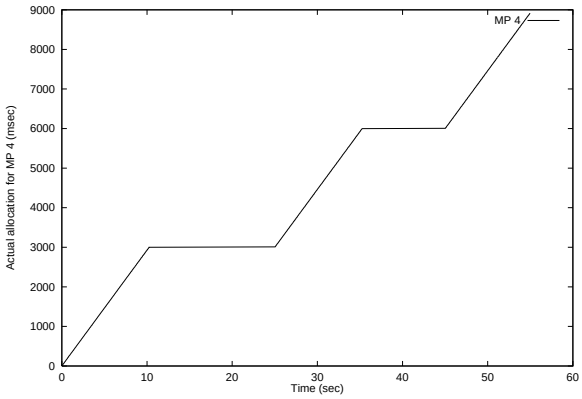
5.1 Scheduling schemes

Several scheduling schemes [11, 9, 10] have proposed providing modular control by statically separating scheduling policies for different classes of applications. The policies are combined using priorities or proportional sharing. The scheduling scheme in [9] allows threads in the hierarchy to define their own scheduling policies within the subtree under the thread. This results in a very flexible and decentralized scheduling mechanism where different entities contribute to the overall scheduling scheme. The scheduling scheme described in this paper is a centralized one as it enables a host to monitor and control external mobile programs more effectively. Further, the above schemes rely on a single scheduler servicing both real-time and conventional applications. This results in a static scheduling hierarchy that is primarily based on different classes of applications. Our scheme, on the other hand, is adaptive, since it allows the scheduling graph to change dynamically.

Many commercial systems [23] provide fixed priority scheduling for real-time applications to combine scheduling

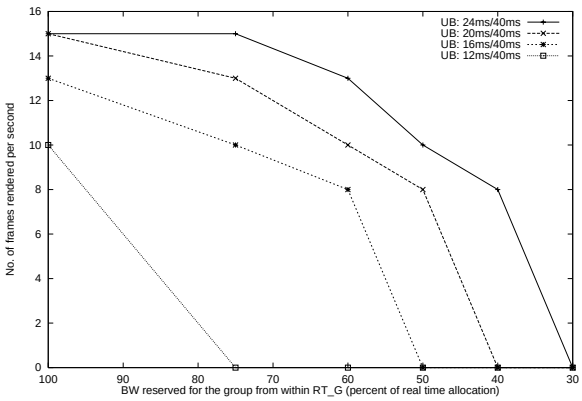


(a) Relative allocation of mobile programs

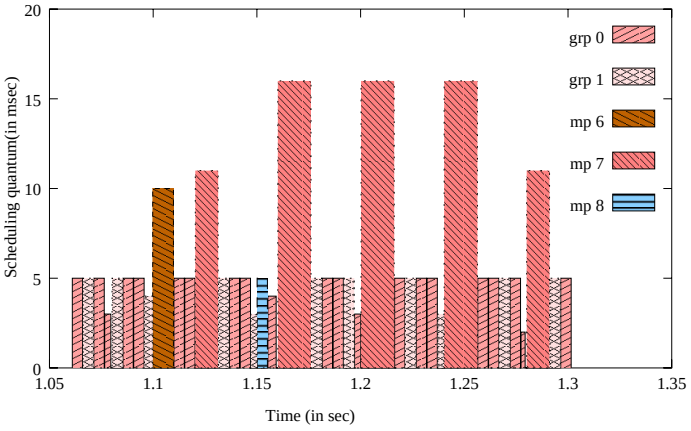


(b) Actual allocation for mobile program 4

Figure 9. Results for lifetime constraints



(a) No. of frames rendered/sec as a function of bandwidth reservation



(b) Scheduling for a combination of real time and non real-time programs

Figure 10. Experiments for real time constraints

of real-time applications with conventional tasks. The problem with these schemes is that they starve the non real-time applications while not providing any guarantees to the real-time tasks. Other systems provide timely execution of real-time tasks on the basis of hierarchical partitioning [9, 10]. However, these schemes are not based on deadlines and do not provide real-time guarantees. Some schemes [17] have implemented deadline based schemes, but they do not provide any guarantees or resource reservations.

In our scheme, the non real-time scheduling algorithm extends the scheduling algorithm used in SMART [17]. We enforce share and priority based constraints over a hierarchical scheduling graph. Real-time scheduling in our scheme is based on Rialto[12] which provides for deadline based resource reservations and guarantees. Our schemes extends the Rialto scheme with upper bounds on the CPU time available to real-time applications.

The notion of upper bound constraints has been studied in several forms. Many version of the Unix operating system provide system calls (e.g., `setrlimit`) for specifying limits on resource consumption. VINO [19], an extensible operating system, provides similar control over allocation of resources. The scheme in our approach supports an adaptive and fine-grained control more suited for the mobile programming environment.

5.2 Mobile programming systems

CPU resource control schemes have been proposed for mobile programs [22, 4, 5]. These systems propose solutions for effective utilizations of resources by mobile programs. In these systems, client and server resource usage constraints are not defined directly in terms of lower bound, upper bound, shares etc. Instead, allocation of resources is based on an economic model. In these models, hosts set prices on consumption of resources, and mobile programs use some form of currency to buy the usage of resources. A host, thus, allocates resources to a mobile program based on the program's ability to buy these resources.

While such schemes can be used to enforce lifetime constraints, a mobile program can cause denial of service attacks if it wealthy. More importantly, since the cost of resources is set uniformly for all mobile programs, it is difficult to define policies in which a host can control allocation of resources on the basis of its preferences or trust relationships. Our approach differs in both the mechanisms used for specifying and enforcing policies. We believe that the economic model can be easily modeled in terms of upper bounds lower bound, shares and priority constraints.

Jres[8] is a scheme for controlling allocation of different kinds of resources (CPU, memory etc) within the Java runtime system. Jres uses binary editing to enforce simple upper bound constraints on Java programs. Our scheme differs from Jres model in that our scheme not only enforces upper bound constraints, but also performs CPU scheduling based on other constraints. We have not used binary editing but implemented the scheme by changing the scheduler within the JVM.

6. Summary

In this paper, we have presented a CPU scheduling scheme that addresses the security and quality of service requirements of a host. The scheme presents an environment for specifying resource usage constraints. Mobile programs specify shares, priority and deadline constraints. Hosts specify shares, priority, upper bound and lifetime constraints. The scheme constructs a scheduling hierarchy to apply a set of algorithms that enforce the various constraints. The non-real time algorithm enforces share and priority based constraints. The real time algorithm enforces deadline constraints. The upper bounds algorithm enforces security constraints specified by the host. Any conflicts between the client and server constraints are resolved by our algorithm composition policy that always favors the server constraints.

We plan to extend our current work in several directions. The first is to extend the scheme so that it can be applied to user-defined resources. We also intend to make the scheme extensible so that arbitrary resource usage constraints, their scheduling algorithms, and algorithm composition policies can be composed on the fly.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] K. M. Baumgartner and B. W. Wah. Computer Scheduling Algorithms: Past, Present, and Future. *Information Sciences*, 57-58:319-345, 1991.

[3] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The World-Wide Web. *CACM*, 37(8):76–82, August 1994.

[4] J. Bredin, D. Kotz, and D. Rus. Market-Based Resource Control for Mobile Agents. In *AGENTS '98, Proceedings of the second international conference on Autonomous agents*, pages 197–204, May 1998.

[5] J. Bredin, D. Kotz, and D. Rus. Economic markets as a means of open mobile-agent systems. In *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3) at Autonomous Agents '99"*, May 1999.

[6] T. Casevant and J. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions of Software Engineering*, 14:141-154, 1988.

[7] D. Chess, C. Harrison, and A. Kreshenbaum. Mobile Agents: Are They a Good Idea? In *Second International Workshop on Mobile Object Systems: Towards the Programmable Internet.*, pages 25–47. Springer-Verlag, 1996.

[8] G. Czajkowski and T. von Eicken. Jres: a resource accounting interface for java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, 1998.

[9] B. Ford and S. Susarala. CPU Inheritance Scheduling. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, Oct. 1996.

[10] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, Oct. 1996.

[11] G. J. Henry. Fair Share Scheduler. *AT&T Bell Laboratories Tech. Journal*, Oct. 1984.

[12] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *16th ACM Symposium on Operating Systems Principles*, Oct. 1997. St. Malo, France.

[13] S. Keshav. *Congestion Control in Computer Networks*. Phd thesis, U. C. Berkeley, 1991.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.

[15] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1), Jan. 1973.

[16] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[17] J. Nieh and M. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. *16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[18] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, Aug. 1992.

[19] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, 1996.

[20] J. Stamos and D. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[21] R. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3), Sept. 1997.

[22] C. F. Tschudin. Open Resource Allocation for Mobile Code. In *First International Workshop on Mobile Agents, MA'97 Berlin*. Springer-Verlag, Apr. 1997.

[23] U. Vahalia. *UNIX Internals, The New Frontiers*. Prentice Hall.

[24] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional Share Resource Management. In *Proceedings of the USENIX 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, Nov. 1994.

[25] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional Share Resource Management. Technical report, MIT Laboratory for Computer Science, June 1995.