Secure Execution of Mobile Programs [*][†]

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

Abstract

There is increasing interest in computing models that support extensibility of systems through code migration. Although appealing both from the system design and extensibility points of view, extensible systems are vulnerable to an external program's aberrant execution behaviors. In this paper, we examine the problems of resource access control and resource consumption. We propose solutions for these problems and analyze their effectiveness.

1 Introduction

On today's Internet, users routinely and often unknowingly download and run programs, such as Java applets. Some web servers permit users to upload foreign programs and execute them. These practices already have the sanction of widespread use, but their security implications have not yet been systematically addressed. In the brief, dynamic history of the Internet, this situation is not unusual. The implementation of new communication mechanisms and computing paradigms has often preceded a rigorous analysis of the security issues they engender.

In computing models that support program migration, a host provides a set of services to an *external program* by loading it and executing it within a local execution environment such as an operating system or a run time system. We call such computing models *extensible computing models*. Two examples are extensible operating systems and mobile programming systems.

Extensible operating systems allow the collocation of external programs by loading them directly into the kernel's address space. Thus, the external program shares its address space with the kernel and all other loaded external programs. SPIN [2] is an example of an extensible operating system that lets users download user-level extensions into the kernel.

Mobile programming systems also support migration by letting users upload programs to a remote host. Further, external programs can stop in mid-execution and then migrate to another host while retaining their state and data.

In addition, Web browsers such as Netscape support extensibility by letting applets be downloaded and executed within the browser. All of these systems provide an execution environment that loads externally defined user programs and executes them within its local name space. We refer to these execution environments as runtime systems.

The extensible computing model is appealing, first, because it lets operating system kernels implement only basic, core functionality, which can then be extended through external programs. This facilitates customization and efficient implementation of specific services and policies, such as application specific memory management or caching policies.

Second, external programs are sometimes far more efficient at utilizing network bandwidth than traditional programming paradigms, such as remote procedure call (RPC). For example, if the external program encodes an application that must filter huge amounts of data, it can migrate to the host with the data, execute there, and then return with its results to the originating host. In this case, the network load incurred by migrating the external program is insignificant

[*] 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.

[†] Portions reprinted, with permission, from (IEEE Internet Computing; volume 2, number 6, pages 35-45; Nov/Dec 1998. ©1998 IEEE), from (13th Conference on Object-Oriented Programming. ECOOP'99; June 1999; published in Lecture Notes in Computer Science number 1628, pages 449-473 by Springer-Verlag) and from (Proceedings of the First International Symposium on Agent Systems and Applications; ©Oct. 1999 IEEE)

compared with the cost of migrating data and processing it locally.

Although appealing both from system design and extensibility points of view, runtime systems are extremely vulnerable to misbehaving external programs. Since external programs run within the same name space as the runtime system, many of the traditional protection mechanisms, such as address space containment, no longer apply. As a result, an external program can maliciously disrupt the execution behavior of extensible systems by interfering with the runtime system's execution or with the execution of other programs within the name space, by using unauthorized resources, by over-using resources. It might also access unauthorized resources or use more than its fair share of resources, denying them to other programs.

In this paper, we focus on two problems: resource access control and resource consumption control. Note that our focus is protecting a runtime system against external programs. We do not address the problem of protecting the communication medium [14] or protecting an external program from runtime systems. Furthermore, we do not address the problem of correctly identifying the source of a mobile program (authentication). The rest of this paper is organized as follows: In Section 2, we describe the two security problems in detail. In Section 3, we present our solution for resource access control. We then describe our approach to resource consumption control in Section 4. We provide a brief summary in Section 5.

2 Security Problems

To help classify security issues, we distinguish two types of resources that a runtime system provides to external programs:

- System resources are those implicitly allocated to external programs, such as memory and CPU.
- Conceptual resources are those explicitly defined and managed by a host. They have well-defined interfaces that external programs use to access resources and request services. For example, a host might provide an interface to a database repository.

This distinction highlights fundamental differences in the mechanisms used to control resource access:

- Because system resources are implicitly allocated and managed by runtime systems, the mechanisms for access control are usually implemented within the runtime system. They also depend on the resource model that runtime systems create. For example, CPU resource access control is traditionally implemented in runtime systems through CPU scheduling algorithms.
- Because conceptual resources are accessed by explicit calls from external programs, access control is generally based on trapping these calls in software. For example, in the Java runtime system (JRTS) [5] calls to protected resources are trapped when these resources call a reference monitor, which the host uses to track and control accesses.

2.1 Resource-centric Security Problems

Security problems result out of conflicts between how a host wants external programs to access its resources and how the external programs access them. There are two kinds of resource access constraints:

- *Access control* refers to restricting a program's access to resources. We focus on limiting access to conceptual resources with explicit interfaces.
- *Consumption control* refers to restricting how much of a given resource a program can consume.

Our primary focus here is on those aspects of resource access and consumption control that are specific to the extensible computing model.

2.2 Resource Access Control

Access control prevents an external program from using unauthorized resources. Because external programs execute within the runtime system's name space, they can directly access any resource by naming it. Naming involves getting a resource handle and using it to invoke operations on the resource. An external program can use the handle to read sensitive files and send the information to remote hosts by accessing network resources. It can also disrupt the operations of a computer system by accessing local resources in an unintended manner. For example, the "Ghost of Zealand" Java applet misuses the ability to write to the screen: It turns areas of the desktop white, making the machine practically useless until it is rebooted. [1]

Once downloaded on a machine, external programs can read private files from local disks or copy proprietary information by accessing databases. Indeed, Hamburg's Chaos Computer Club demonstrated on German television how to use ActiveX, Microsoft's external programming system, to steal funds. In this exploit, the victim uses Internet Explorer to visit a Web page that downloads an ActiveX control. The control checks to see if Quicken, a financial management software, is installed. If it is, the control adds a monetary transfer order to Quicken's batch of transfer orders. When

[1]For full details, see http://www.finjan.com/applet_alert.cfm or http://www.internetworld.com/print/1998/05/11/webdev/19980511-hostile.html

the victim next pays the bills, the additional transfer order is performed. All of this goes unnoticed by the victim.[2]

Although the resource access control problem has been studied within the context of traditional systems, the problems are different for extensible systems [6] in several ways.

First, in extensible systems, authorization is more complex. In traditional operating systems, programs run on behalf of principals who are given certain rights. Once a program attains these rights, they usually remain valid during the program's execution. In extensible systems, however, an external program contains individual components that might have different rights and permissions. Hence, the level of granularity at which access rights must be checked and enforced is much finer – sometimes at the level of individual objects and functions.

Second, security mechanisms should be independent of the site's resources. Most traditional operating systems manage a fixed set of resources such as memory, CPU, files, and the network. Extensible systems, however, must manage resources that can vary from site to site.

Finally, most traditional operating systems implement an access control model that either allows or denies access. Extensible systems can allow conditional resource access based on runtime or program state [10]. For example, a database vendor can specify that if there are more than 20 external programs in the system, each external program can only access its database up to 10 times.

2.3 Resource Allocation Problem

When external programs use more than their share of resources, they leave the system vulnerable to attack. For example, external programs can stage denial-of-service attacks by intentionally over-using CPU, thereby denying CPU to other programs and the runtime system. Consumption control is required not only for system resources, but also for conceptual resources. For example, by opening multiple socket connections and flooding the network with data, an external program can deny network resources to other programs.

The resource-consumption problem is similar to the CPU-scheduling problem in that both require the runtime system to control the resources allocated to requesting programs. The two problems differ in the type of control that runtime systems need to exercise. The primary goal in CPU scheduling algorithms is to allocate CPU so it provides some quality of service (QoS) to executing programs. The QoS requirements can be specified as constraints, such as optimal resource utilization, response time, lower bounds on resource allocation, and deadline. In mobile systems, we can 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.

Notation: Let the term $\sigma_t(P)$ denote the amount of CPU allocated to a program P in a *schedulable unit*[3] t.

2.3.1 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.

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

2.3.2 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

[2]For more on this, see http://www.iks-jena.de/mitarb/lutz/security/activex.hip97.html or http://www.iksjena.de/mitarb/lutz/security/activex.en.html

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

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 [1], 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 [3, 13], 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*.

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.

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

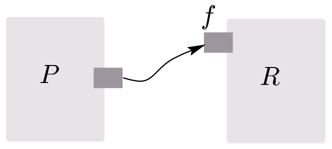
  A host can also specify upper bounds in absolute form.

- **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.

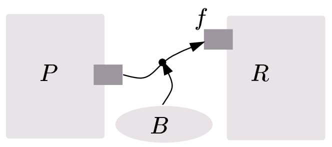
2.3.3 The Scheduling Problem The CPU resource control problem is, therefore, one of developing a scheduling scheme that, given client and server resource usage constraints, constructs a schedule that enforces the constraints.

3 Access Control Model

Out access control model contains two parts: a resource model for representing resources and an access constraint specification language. We describe the two in detail below.



(a) Default method invocation semantic



(b) Security constraints on method invocations

Figure 1. Method invocation semantics

3.1 Resource Model

A site provides many resources to a mobile program. These resources include classes for utility libraries, accessing files, networks, and interfaces to other resources such as a proprietary database. For instance, a site providing access to a weather database exports a set of interfaces that specify how the database can be accessed. In our security model, each Java class or method represents a resource and, thus, is a unit of protection. Our access control mechanism does not differentiate between system classes and user-defined classes, or between locally defined classes and classes downloaded from remote hosts. The model also allows the definition of class-subclass relationships among resources using the Java's inheritance model.

3.2 Access Constraint Specification Language

The access constraint specification language contains two parts: a notation for constraining accesses to resources and its inheritance model.

3.2.1 Access Constraints We first describe the motivation behind our access control language. A Java program uses a resource by invoking its methods. In Fig. 1(a), we show that program P invokes a method f to access resource R. During an execution of P, the control jumps to f, executes f, and returns back to P upon termination. The Java compiler implements a simple access semantics in which there are no constraints on accesses to R through f.

Our approach is to allow a host to make the access relationship between P and R *conditional* by adding a constraint, B (see Fig. 1(b)). The access constraint is specified *separately* from both P and R and has the effect of imposing the constraint that P can invoke f on R only if condition B is true. A site, thus, restricts accesses to specific resources by enumerating a set of access constraints, which forms a site's access control policy.

Below, we present the core aspects of the language. The following EBNF shows how a site can specify access constraints:

```
Constraints     ::= { AccessConstraint | EnableStatement }
AccessConstraint::= deny '('[Entity]Relationship Entity')'
                        [when Condition]
EnableStatement ::= enable '(' Entity
                        [Relationship Entity] ')'
Relationship    ::= ↦  | ⊣
Entity          ::= ClassIdentifier | MethodIdentifier
                        | GroupName
Condition       ::= BooleanExpression
GroupStatement  ::= define group GroupName
                        '{' Entity ';' { Entity } '}'
```

A site controls accesses to different resources (Java objects) by defining a set of AccessConstraints. We describe the various terms in the grammar informally below:

- Entity: An entity denotes objects and method invocations of Java programs. A ClassIdentifier, thus, identifies the set of objects to which a given access relationship applies. Similarly, a MethodIdentifier denotes a set of invocations of a method. In addition, an entity can be a group of entities.

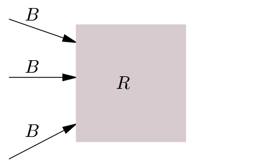
- Relationship: The composition mechanisms of a programming language allow one to define various relationships (data composition through aggregation and inheritance, and program composition through method invocations) among the entities of a program. We are primarily interested in the following two access relationships here:

  1. Instantiate ($\dashv$): A relation $E \dashv R$ exists if an entity E creates an instance of class R.
  2. Invoke ($\mapsto$): A relation $E \mapsto R$ exists if an entity E invokes an entity R.

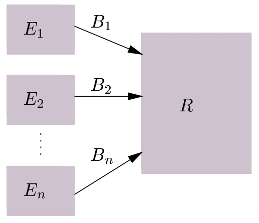
- Condition: The term Condition denotes a boolean expression that can be defined in terms of object states, program state (global state), runtime system state, security state, and parameters of methods.

- Enable: In addition to access constraints, we support an enable statement that allows a host to certain accesses, thereby overriding these constraints. This is needed in cases when a host wants to override the default principles of least privileges. For example, assume that a security policy specifies that an applet cannot access the file system. The security infrastructure implements a default policy of least privilege, which ensures that the applet cannot access the file system directly or indirectly by calling other methods that access the file system. However, in many cases, this may not be desirable [15]. For instance, suppose the applet can write to the screen using the font files stored on the disk. In such cases, we want to enable the display manager to be able to access these files, regardless of the calling program. The `enable` statement allows one to override the default policy. This is similar to the `enablePrivileged` command of the JDK1.2 security model [5].

- Group: The current implementation defines a group based on its name. However, this can be extended to define an entity on the basis of its source, signature, or behavior pattern.



(a) Global constraints



(b) Selective access constraints

Figure 2. Category of access constraints

3.2.2 Semantics

An access constraint of the form

deny (E ξ R) when Condition

specifies that entity E cannot access R through relationship ξ if Condition is true. Since E is optional, there are two kinds of access constraints: *global constraints* and *selective access constraints*. Global constraints denote those constraints that do not depend on the initiator of the access relationship. As shown in Fig. 2(a), no program can access R when B is true. For example, a host may specify the constraint that no Java code can access a set of proprietary files.

Selective access constraints denote those constraints that depend on the initiator of the access relationship. For instance, as shown in Fig. 2(b), each entity E_i's access to R is constrained by a separate and possibly different B_i. A site can use selective access constraints to associate different security policies with different Java programs that come from different sites.

3.2.3 Inheritance of Access Constraints We now present an inheritance model that describes what the denial of a resource means to its subclasses.

Assume that a site defines two resources, R_c and R_s:

```
class Rc {
    public void f();
    public void g();
    public void h();
}

class Rs extends Rc {
    ⋮
}
```

R_s is a subclass of R_c. R_s inherits methods f, g, and h from R_c. Assume that the site defines the following constraints on the resources:

deny ($E \mapsto R_c.f$) when B_{cf}
deny ($E \mapsto R_c.g$) when B_{cg}
deny ($E \mapsto R_s.f$) when B_{sf}
deny ($E \mapsto R_s.h$) when B_{sh}

There are two components to the inheritance model:

- Inheritance of access constraints: A subclass inherits all access constraints from its superclasses. Hence, the resulting access constraint on invocations of g on an instance of R_s is defined by the following expression:

  deny ($E \mapsto R_s.g$) when B_{cg}

  Access constraints are not inherited from subclasses to superclasses. Hence, although the access constraint on h in R_s is B_{sh}, there are no access constraints on h in R_c.

- Strengthening of access constraints: A subclass cannot override its inherited constraints. Specification of additional constraints in the subclass only strengthen the constraints defined in its superclasses. Hence, the resulting access constraint on invocations of f on an instance of R_s is:

  deny ($E \mapsto R_s.f$) when $B_{cf} \vee B_{sf}$

  In other words, method $R_s.f$ cannot be invoked from E if either B_{cf} or B_{sf} is true.

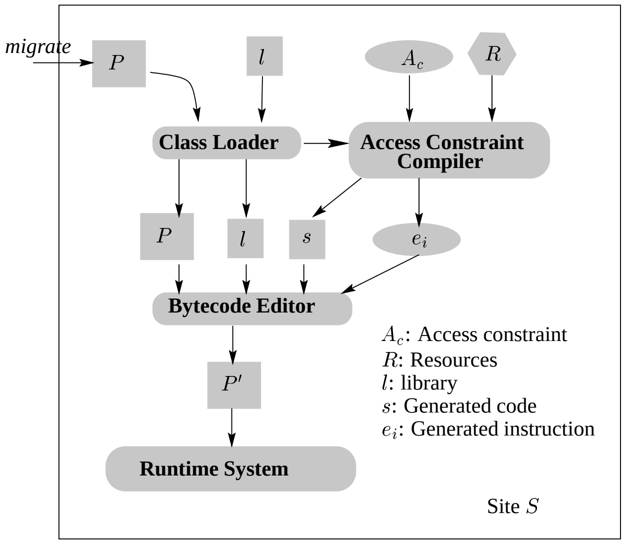


Figure 3. Security policy enforcement of mobile programs

This model of inheritance ensures that a mobile program cannot override access constraints on methods by defining a subclass and by weakening the access constraints. Also, the above inheritance model applies for access constraints on $\dashv$ as well. That is, if a class R_c cannot be instantiated, none of its subclasses can be instantiated.

3.3 Access Constraint Enforcement

An enforcement of access constraints on a resource involves placing interposition code between the resource access code and resource definition code. The interposition code checks if a specific resource access is allowed. It can be inserted *manually* by site managers, generated by the compiler, or defined by the runtime systems or operating systems through special system calls. For instance, in the Java runtime system [4, 5], resource developers manually insert calls to a reference monitor in the resources they want to protect. The reference monitor consults access control policies to check if a specific resource access is allowed.

We use an alternate approach for generating interposition code. In this approach, a set of tools generates the interposition code and integrates them within mobile programs and resources before they are loaded in the JVM. In this approach, there are no reference monitors. In essence, the approach generates reference monitors on the fly and integrates them within the relevant Java programs and resources. The approach, thus, eliminates the need to manually include calls to reference monitors in resource definitions.

In Fig. 3, we describe our implementation for enforcing access control policies on Java programs. We show a Java program P that migrates to a site S. R denotes resources that the site makes available to mobile programs; and l denotes local libraries linked into P.

During class name resolution and dynamic linking, the Java class loader [8] retrieves R and l and passes them to a tool, called the *access constraint compiler*. The access constraint compiler examines P, R, and l to determine the resource access relationships that must be constrained in order to implement the access constraint A_c. It then generates interposition code s that implement the specific access constraints. It also generates a set of editing instructions e_i for the bytecode editor. The bytecode editor uses e_i to integrate s within P, R and l. The transformed programs and resources are then loaded into the JVM and executed.

3.4 Code Generation and Binary Editing

We now describe the nature of the code that is generated and its integration within mobile programs. Our code generation and editing involves modifying class definitions in order to add runtime state to classes and to insert runtime checks into methods.

An access constraint of the form

```
deny (E ξ R) when B
```

is implemented by generating the following code:

```
if (B)
    then error(); // raise exception
else
    access R
```

and patching it into classes and methods. The nature of the editing depends on the nature of the access constraints. Global constraints of the form

```
deny (ξ R) when B
```

specify constraints on accesses to R without any regard to objects or methods that may access R. The generated code is, thus, integrated into the methods of R. On the other hand, selective access constraints of the form

```
deny (E ξ R) when B
```

imposes conditions on accesses to R from E. In this case, R must determine if it had been called by E. We implement this using stack inspection [15, 16], which we will discuss in the following sections.

We also support addition of security states to specific Java classes in order to monitor site-specific behavior. This mechanism allows a site to customize its security policies, especially if the policies cannot be represented directly by the policy language. Security state objects are added to a class definition by using the statement:

```
add SecurityStateType SecurityStateObject
    to R
```

The constraint compiler generates code for initializing these objects.

3.5 Discussion

In this section, we analyze the proposed technique for its suitability as an access constraint enforcement mechanism and discuss its performance behavior.

3.5.1 Characteristics of the Approach In our approach, a site specifies access constraints separately from mobile programs, resources, and other class definitions. Further, the access constraint enforcement mechanism is not part of either the Java runtime system or the compiler. This impacts how access control code is managed and enforced at a site:

- Both access constraints and resource definitions can be modified independently. This makes it easy for a site to specify different access constraints for different mobile programs for the same resource. For instance, a site may specify that mobile program P can access R under condition B_p whereas mobile program Q can access R under condition B_q.

- The same set of access constraints can be applied to different resources without requiring one to copy it from one resource to another. For example, if a single access constraint B applies to multiple resources, it can be defined once and used for all resources.

- An important advantage of the separation is that our approach can be used for enforcing security on resources that were not designed with security in the first place. In other words, the security component can be added to a resource after it has been designed and implemented. Thus, it frees a library or resource designer from worrying about security concerns when designing and implementing the library.

Despite these advantages, there are several limitations of our approach.

3.5.2 Performance Analysis In this section, we describe the performance behavior of the access constraint enforcement mechanism. Specifically, we analyze the following:

- What are the time and space overheads associated with our approach?
- How does our approach perform with respect to the Java runtime system's approach for enforcing access control?

We performed our experiments on a 266 MHz Pentium II running Red Hat Linux 5.0. The results show that both the time and space overheads of the approach are moderate. Further, the approach performs better than the Java runtime system in certain cases.

Overhead Measurements: We measured both the time and space costs of modifying resources.

There are four factors that affect the execution time associated with access constraint check code generation and editing:

- the cost associated with reading a method
- the number of access constraints
- the types of constraints
- the number of occurrences of restricted methods in a program

We do not consider the cost of reading class files in our measurements since the run-time system must perform this operation anyway.

In the first experiment, we looked at how the size of the method being modified affects the cost of editing. In this experiment, only a single method invocation must be wrapped. The cost of editing here is minimally affected by the size of the method. The cost varied between 0.08 and 0.16 seconds for methods ranging from 0 to 3200 instructions. In the second experiment, we looked at how the cost of editing changes when the number of method calls that needs to be wrapped changes. We found the cost to be proportional to number of methods that are wrapped.

We have also calculated the increase in size caused by adding code to class definitions. While the amount of code that is added to a class is independent of the size of the class, it depends on the number of method invocations that need to be wrapped and the complexity of the boolean portion of the constraint. For one wrapper, the minimum addition size (for a true boolean constraint), is 56 bytes. For two simple boolean expressions, it is about 206 bytes.

Performance Comparison: We now compare the performance behavior of our approach with the runtime system approach, as implemented in the JDK 1.1.3.

For this experiment we created a small program to test the performance of implementing security checks around one method invocation. Since the actual amount of work a

```
class SecState {
  public SecState() {count = 0;}
  public int check()
    { count++; return count; }
  private int count;
}
```

(a) Security object

```
add SecState SecurityState to R
deny  ↦  R.f()V when
  #1.SecurityState.check() > 1000000
```

(b) Control access constraints

Figure 4. The binary editing approach

particular site must perform depends on both the complexity of the access control policy and the number of restricted method invocations in a program, implementing a single policy statement once forms a good basis for comparison. We based our comparisons on an access control policy limiting the number of times a particular resource can be accessed. The complete code for our approach is shown in Fig. 4. We implemented the same policy using Java's security manager as shown in Fig. 5. The test program calls the constrained method variable number of times. The access policy is that the method cannot be called more than 1000000 times.

Figure 6 shows the execution times of our approach and the Java's runtime system approach. In our approach, there is an initial overhead of about 0.08 seconds for code editing, which does not occur in the Java runtime system. However, after about 100000 method calls, our approach performs better than the Java runtime system. This is because our approach inlines the access control check code, whereas in case of the Java runtime system approach, each access constraint check involves making two method calls: one to the system, to get the security manager, and another to the security manager itself. We can reduce our cost even further by pre-editing the methods if we know that only a single access constraint will be applied to the method, as is the case in the Java runtime system approach. Our approach, in this case, will then always outperform the Java runtime system approach.

In the second experiment, we ran the same program with no policy implemented. As shown in Fig. 7, the Java runtime system is always less efficient that our approach. This is because in the Java runtime system approach, a method must always call the runtime system to check if there is a se-

```
class newSecMan
  extends SecurityManager {
  public newSecMan() {count = 0;}
  public void checkf()
      throws SecurityException {
    count++;
    if (count > 1000000)
      throw new SecurityException();
  }
  int count;
}
```

(a) Security Manager

```
class R {
  public void f() {
    newSecMan security;
    security =
      System.getSecurityManager();
    if (security != null)
      security.checkf();
  }
}
```

(b) Resource definition

Figure 5. The Java Runtime System-based approach

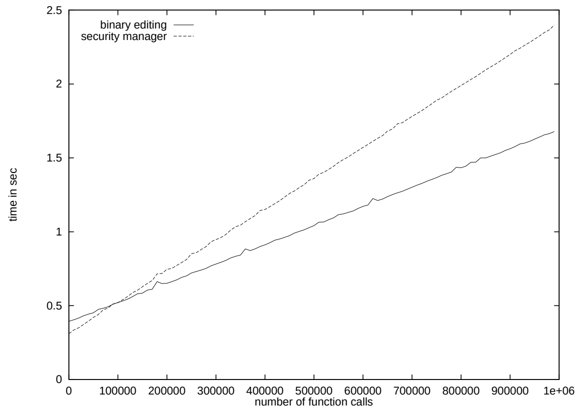


Figure 6. Comparison of execution times with a policy

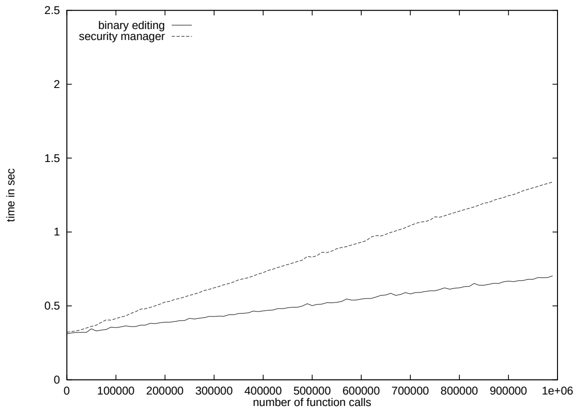


Figure 7. Comparison of execution times without a policy

curity manager installed, incurring the overhead of this call. Our approach does not incur any overhead since it does not add any code to methods that do not need to be constrained.

4 Resource Consumption Control

In this section, we describe our approach to resource consumption control. 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.

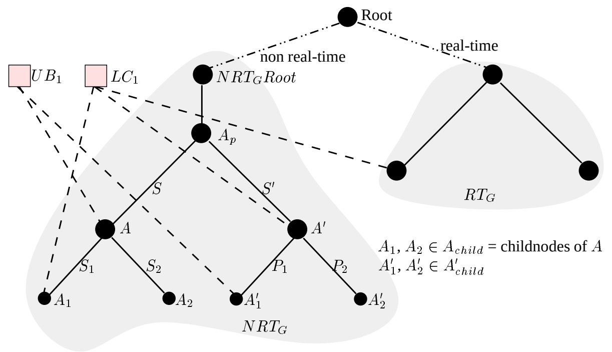


Figure 8. The scheduling graph

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

We now 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.
- 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.

4.1 Construction of Scheduling Graph

The scheduling scheme first builds a *scheduling graph* (Fig. 8) 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 Fig. 8, 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 Fig. 8, 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 Fig. 8 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.

4.2 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

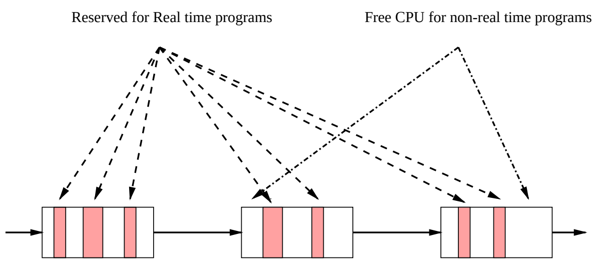

Reserved for Real time programs
Free CPU for non-real time programs

Figure 9. List of quantum time chunks with reservations for real-time programs

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* (Fig. 9). Within each quantum time chunk, it schedules mobile programs from RT_G according to their reservations. The reservations fix the times when the 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 ($NRT_G Root$). The scheme traverses from $NRT_G Root$ to one of the leaves of the graph.

In the next sections we describe the individual algorithms for non real-time and real time programs.

4.2.1 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 [11] 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 (Fig. 8). 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:
$$UVT_A(t) = LVT_{A_p}(t) \quad (1)$$
Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing:
$$UVT_A(t) = UVT_A(\tau) + \frac{t-\tau}{S} \quad (2)$$

- LVT: The lower virtual time at A selects one of A's children. Initially, when A joins the hierarchy,
$$LVT_A(t) = 0 \quad (3)$$
Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing,
$$LVT_A(t) = LVT_A(\tau) + \frac{t-\tau}{\sum\limits_{a \in A_{child}} S_a} \quad (4)$$

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:
$$VFT_A(t) = UVT_A(t) + \frac{Q}{S} \quad (5)$$

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:

$$VFT_A(t) = VFT_A(\tau) + \frac{Q}{S} \quad (6)$$

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 (Fig. 8). 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.

4.2.2 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 [7]. 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*[12]. 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 (Fig. 9) 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 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) [9] order. Using EDF for adding new reservations increases the number of reservations satisfied.

In Fig. 10, 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

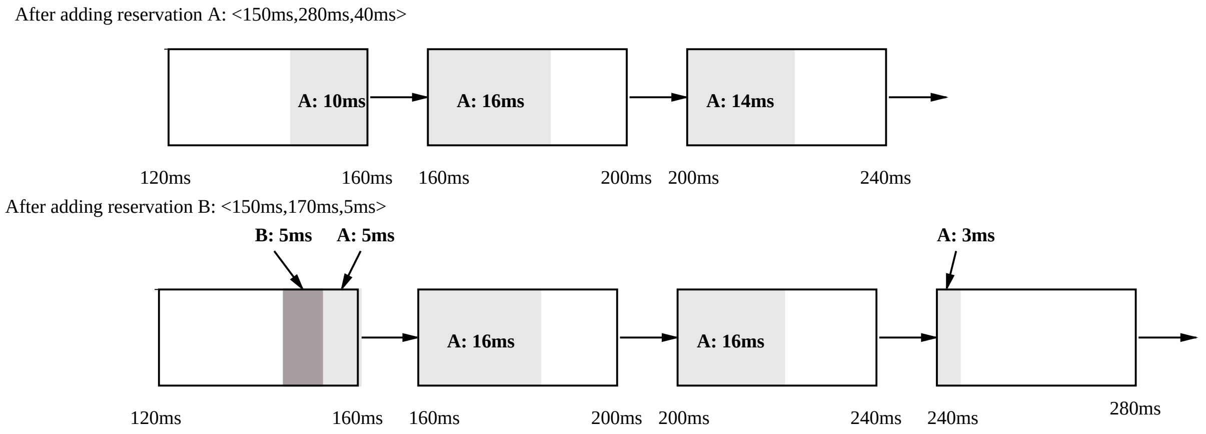


Figure 10. List of quantum time chunks for two reservations

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.

4.2.3 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 Fig. 8. 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.

4.3 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).

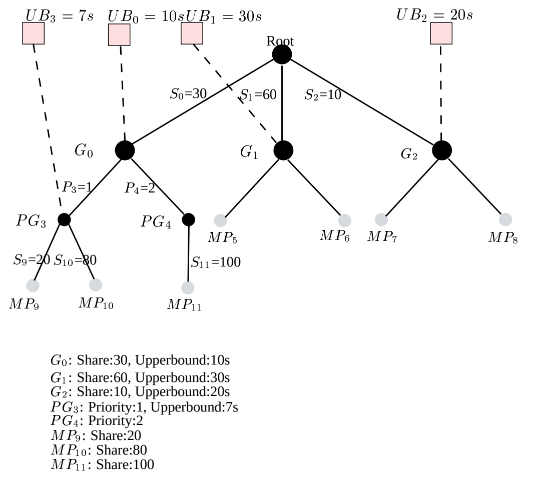
We conducted several experiments on the simulation engine and the JVM. The goals of these experiments were to examine the effectiveness of the scheme in (i) satisfying both real-time and non real-time constraints; (ii) enforcing upper bound and lifetime constraints; and (iii) satisfying constraints that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

4.3.1 General scheduling behavior

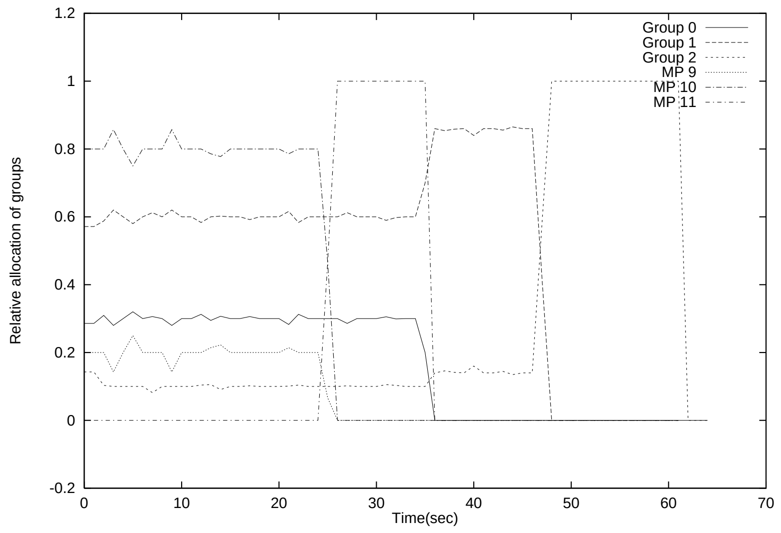
The first experiment demonstrates how the scheme schedules groups of mobile programs that are constrained by shares, priorities and upper bounds. Further, it shows how the upper bound constraints interact with shares and priority constraints. We performed this experiment on the modified JVM. In Fig. 11(a), we show the hierarchy constructed from the client and host resource usage constraints. In Fig. 11(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



(a) The scheduling graph



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

Figure 11. General scheduling behavior of the scheme

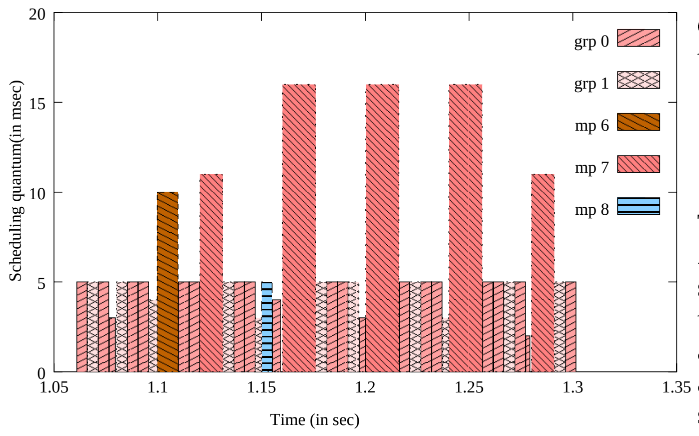


Figure 12. Scheduling for a combination of real-time and non real-time programs

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.

4.3.2 Real-time Programs

In this experiment (Fig. 12), we demonstrate how the scheduling of real-time programs takes place in the presence of non-real time programs. We conducted the experiments within the simulation engine.

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 Summary

In this paper, we have presented schemes for enforcing both resource access and resource usage control.

We have described a mechanism for implementing general access control policies on mobile programs. There are two components of our approach. The first is a simple declarative access constraint language that allows a site to restrict accesses to the objects and methods of the system. The declarative nature of the language makes it easy to specify policies while still allowing a hook to express procedural policies if necessary. The second is a set of tools that enforce the specified constraints by editing mobile programs and resources. Our approach's appeal is that a site can specify access constraints separately from both mobile program definitions and resource definitions. This separation of concerns has a number of benefits. Both access constraints and resource definitions can be modified independently. Sites can easily specify different access constraints for different mobile programs for the same resource. Finally, our approach can enforce security on systems that were not originally designed with security in mind.

We control usage of CPU by developing 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.

References

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

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO., December 1995.

[3] 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.

[4] L. Gong. Java security: Present and near furture. *IEEE Micro*, 17(3):14–19, May-June 1997.

[5] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997. Available to Usenix members at http://www.usenix.org/publications/library/proceedings/usits97/gong.html.

[6] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, Jan. 1998.

[7] 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*, October 1997. St. Malo, France.

[8] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 10, Vancouver, October 1998. ACM.

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

[10] D. V. Miller and R. W. Baldwin. Access control by boolean expression evaluation. In *Fifth Annual Computer Security Applications Conference*, pages 131–139, Tucson, AZ, 1990. IEEE, IEEE Comput. Soc. Press.

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

[12] 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, August 1992.

[13] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3), September 1997.

[14] J. Vitek, M. Serrano, and D. Thanos. Security and communication in mobile object systems. In *Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MOS '96*, number 1222 in Lecture Notes in Computer Science, pages 177–199, Linz, Austria, July 1997.

[15] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architecture for Java. In *16th ACM Symposium on Operating Systems Priciples*, volume 31(5) of *Operating System Review*, pages 116–128, Saint Malo, France, Oct. 1997.

[16] D. S. Wallach and E. W. Felton. Understanding Java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998. IEEE, IEEE Comput. Soc.