# Dynamically Controlling Resource Usage using Resource Control Lists


Akihiko Miyoshi

*Department of Electrical & Computer Engineering*

*Carnegie Mellon University*


Submitted in partial fulfillment of the requirements for

the degree of Master of Science in Electrical and Computer Engineering.

## Contents

## Abstract

Resource reservation has now become a well-accepted technique in OS support for QoS provisioning. Promising results from the first generation of these systems implementing resource reservation lead us to believe that this technique will become widely used in general-purpose systems such as Linux. In this paper, we first identify unaddressed issues in the first generation of reservation-based systems. Specifically, they constitute a rather vulnerable infrastructure with the possibility of uncontrolled usage of resources. This can potentially lead to serious DoS (denial of service) attacks due to the lack of strict access control and a way to specify and enforce a system-wide policy. Also, they are not able to capture the dynamic nature of the applications that use system resources.

In this paper, we propose and evaluate the use of a novel first-class abstraction called Resource Control Lists (RCLs) for use in reservation-based operating systems. In general, an RCL describes "*who* is allowed to use *what* resources, *how much* and for *how long*". This new facility allows the specification and enforcement of a flexible system-wide policy of resource usage, and of allocating resource reservations. We further enhance dynamic resource control by introducing three mechanisms: (1) to change the resource usage policy reacting to dynamic system changes and application state, (2) to dynamically define the user (resource principal) of the resources through **propagation** of reservation, and (3) a **parental control** feature that ensures that the resources consumed by a process and all its forked children are bounded by the parent's reservation. RCLs can be stacked to create multiple levels of resource control. Finally, we have built support for RCLs into Linux/RK [16], CMU's real-time extensions to Linux incorporating Resource Kernel (RK) features. Quantitative measurements show that the overhead of our new features is acceptable while providing flexible functionality.

## 1 Introduction

Over the past few years, the need to support multimedia and other real-time applications motivated OS support for resource reservation and QoS provisioning. Unlike resource management schemes implemented on traditional time-sharing OSs where all applications compete for resources, on systems providing resource reservations, applications receive a promised amount of resources such as CPU time, or disk I/O and network bandwidth by making *reservations*. This enables applications to make predictable progress under arbitrary system load. For example, a movie player application without resource reservation may not be able to provide an ideal framerate because of other competing activities demanding resources. This often results in jitters and loss of framerate in the movie. By making reservations, the movie application can provide the desired framerate even under heavy system load.

Several abstractions have been proposed and implemented in different operating systems. For example, Mercer [14] introduced resource reservation in the form of *processor capacity reserves* on RT-Mach [22]. Applications could request and obtain a "reserve" which was then set aside for the exclusive use by an application. The following mechanisms were observed to be necessary to support reservations:

- *Specification of reservation:* A reserve represents a fraction of a processor's cycles and is specified as $C$ units of time available every successive $T$ units of time. This allows application programs to specify their processor and timeliness requirements.
- *Admission control:* An admission control module evaluates reservation requests to decide whether to admit them or not. The admission control policy must take into account the scheduling discipline deployed. For example, if the earliest deadline first scheduling algorithm is used, the system can accept up to 100% of processor capacity. For the rate-monotonic scheduling algorithm, at least 69% of processor capacity can be guaranteed.
- *Scheduling:* The scheduler schedules activities consistent with the admission control policy. It guarantees that once an application is granted a reservation by admission control, then it is assured of the availability of that resource.
- *Accounting/Enforcement:* It is necessary to accurately measure the computation time consumed by each

application to ensure that programs do not overrun their reservations.

More recently, Banga et al. [1] have introduced a notion called *resource containers*, which refines the abstraction of *reserves* by allowing the original *reserves* functionality implemented on a microkernel (RT-Mach) to be implemented on a monolithic operating system. This requires, for example, additional integration with mechanisms to account for kernel-mode processing such as network protocol processing. The accounting of multiple activities in different protection domains is facilitated by an abstraction called the *resource principal*, which is an accountable entity to which the system can charge resource usage for a certain activity. Both the "reserve" and "resource container" abstractions allow the clean separation of protection domain and resource principals.

The above-mentioned (and other related) work have laid the foundation for applications to specify and obtain resource guarantees. However, these first-generation reservation systems have shortcomings that prohibit them from being deployed in dynamic computing environments such as networked servers. By dynamic, we mean an environment where a diverse class of applications, including trusted and untrusted ones, coexists. Unless the system can capture the dynamic behavior of applications and their changing requirements, the full potential of resource reservation cannot be exploited.

Let us consider an example. A program may depend on the progress of other programs to make progress, but *only* at certain times (such as a client in a client-server environment). Then, how do we attach a reservation to that application only at a certain time? Also, the requirements towards resources may change over time (eg. night-time vs. day-time) or by the mere fact that another program exists (eg. when a gold-card member executes a program, resource usage of programs executed by a silver-card member should be limited). Then, one must somehow describe the relationship between the two programs with regards to resource reservation, and also take into account the privilege of the principal who made the request. As we can see, it is extremely difficult if not impossible to describe the behavior or resource requirements of applications by using only the facilities provided by previous reservation-based systems. It is also easily imagined that the programmer for each application will not be able to efficiently embed such behavior into their code.

As [14] points out, there are two structures one should consider when deploying reservations:

1. a system-wide policy for allocating resources based on application-level and system-level QoS requirements, and
2. the abstraction and mechanism for scheduling and controlling those resources.

The second aspect of abstraction and mechanisms for reservation has been studied in depth. However, we believe that the first aspect of having a system-wide resource allocation policy has not been dealt with seriously in a structural manner. To define and enforce application-level and system-level policy requirements, we propose that the following features are critical for use in a dynamic environment:

- Resource access control features to determine which applications can request what resources, for how much, and for how long,
- Dynamic definition of resource principals to deal with client-server systems, and
- Detecting and responding to time-varying and semantic requirements of reservation and resource usage policies.

In this paper, we address the above requirements by introducing a simple but powerful concept called Resource Control Lists (RCLs). RCLs provide a practical bridge between the *mechanisms* controlling the resources (such as scheduling and admission control), and the *policies* for allocating resources based on application-level and system-wide QoS requirements. With the use of RCLs, an administrator/user of the machine obtains more control over resources rather than requiring or letting application programs to embed the resource control policies. For example, the administrator will be able to guarantee or limit the amount of resources granted to certain class of applications by specifying the policy using RCLs. This does not require modification to the actual applications.

We also introduce three other abstractions in order to support the flexible and disciplined use of RCLs:

- The notion of Resource Controllers is used to manage resources at different levels of granularity and to change RCLs dynamically,
- The notion of Events is used to track and communicate information about resource usage between resource controllers, and

- The notion of Propagation is used to modify the user (resource principal) of a resource dynamically.

An innovative feature of our approach is that multiple resource controllers can be stacked on top of one another, enabling RCLs with different capabilities and granularities.

### 1.1 Related Work

There have been various scheduling frameworks to provide QoS provisioning and timeliness guarantees. The most popular scheduling paradigm in commercial real-time operating systems today is perhaps the fixed-priority preemptive scheduling of tasks. In this paradigm, each task (either a thread or process) gets assigned a fixed priority, and a scheduler dispatches for execution the highest priority task ready to run at any given point in time. The popularity of this paradigm has also been bolstered by the availability of analytical frameworks such as Rate-Monotonic Analysis [11] and widespread support for this paradigm in industry and international standards such as POSIX, Ada, RT-CORBA, RT-UML and RT-Java.

Unfortunately, the priority-driven scheduling paradigm has a wide-ranging set of problems associated with it as well. In certain circumstances, other paradigms are more appropriate. These problems include:

1. *Priority Assignment Problems*: The timing constraint of an application must be mapped to a single priority value, whose range and direction changes from one OS to another. Even on a given system, once priorities have been assigned, the introduction of a new task may force priorities to be re-assigned. More importantly, the priority of a task depends on the needs and existence of other tasks, requiring perhaps knowledge of the entire set of applications running on the system. This requirement may become impossible to follow in a dynamic environment (such as on the desktop or within an element of a public network).

2. *Lack of Timing Fault Containment*: One high-priority application that consumes more resources than it should (e.g. by entering into a long, possibly infinite, loop) can endanger the health of the whole system. More subtly, the correctness of a well-tested real-time application can be compromised by the incorrectness of another real-time application. This normally leads to serious reluctance to bringing new applications and/or features.

3. *Lack of Guaranteed Resource Shares*: In many application contexts such as QoS-enabled networking, multimedia servers, Web services and even desktop environments, applications or application groups may have timing constraints and/or the need to obtain weighted shares of available system resources [2, 10]. For example, a higher-paying ISP customer may want guarantees of 40% of a Web server processing, disk and network bandwidth resources to be allocated to its logical web-site hosted by the server [6]. It is difficult to express these requirements in a fixed-priority scheme.

As a promising alternative, resource reservation has been adopted in many QoS-enabled systems. Numerous systems have addressed different aspects of reservation-based systems. A reservation-based approach with explicit admission control and strict enforcement of resource consumption has been proposed by [24, 10, 14, 8, 17, 23]. An early approach to this paradigm can be found in the Spring Kernel work [20] at the University of Massachusetts.

Two core variants of this paradigm have been studied at length in the literature. The first is the weighted fair-share (fine-grained weighted round-robin) scheme typified by the work presented in [2, 10]. The other is roughly based on the Liu and Layland [4] periodic task model with an OS-internal priority-based scheduling policy and enforcement of scheduling policy assumptions. This latter approach is typified by the Resource Kernel approach at CMU [17], the Nemesis project at Cambridge [8], the Open Systems project at UIUC and the work by Buttazzo's group [5]. The processor capacity reservation scheme [14] has been implemented in RT-Mach for admission control and temporal isolation between tasks. Reservation of other resources such as disk I/O and network bandwidth has also been addressed[15, 7, 18].

Jeffay et al. [10] added support for proportional fair-share scheduling in FreeBSD at the University of North Carolina. Rialto [13] provided support for time-driven scheduling with a reservation abstraction. A two-level hierarchical scheduler was designed for an open system environment at the University of Illinois at Urbana Champaign. Significant contribution of these researchers include the evaluation of the mechanisms and scheduling algorithms that provided reservations.

Several other researchers have gone further to address how reservations can be deployed in their systems based on mainstream time-sharing systems. The notion of "resource containers" [1] has been implemented on Digital UNIX. It is pointed out that accountable entities can span multiple protection domains. For example, multiple processes can be a member of a single "resource container" and form a single accountable entity.

Eclipse [3] provides hierarchical proportional resource sharing based on FreeBSD. Eclipse represents a hierarchy of resource reservations using the /reserv file system. In the /reserv file system, applications specify resource reservations as directories in a new file system mounted under /reserv. Reservation is controlled only by using standard APIs for file operations such as *open, close, write*. Applications can create 'sub-reservations' inside a 'reservation' by operating on the files in the directory. Access permissions on the directories represent the access permissions for the reservations. Hence, a program that can access a directory is allowed to access the reservation the directory represents. This approach solves the "*who* is allowed to request a reservation" problem. But neither of the schemes transparently capture the dynamic behavior of applications with both also requiring modifications to applications. This means that, in both "resource container" and "Eclipse", applications had to embed the resource usage policy within themselves, and legacy applications that were not specifically tailored to use reservations never benefited from the services.

More recently, commercial operating systems such as Solaris [21] and AIX [9] have attempted to provide access control facility to resource shares. Both propose to use crons to statically program the system behavior. In both operating systems, the user of the resources (resource principal) cannot be defined in a dynamic manner.

To our knowledge, the power, clarity and conciseness of our RCLs and the dynamic propagation of reservations are not supported on any of the above mentioned systems.

### 1.2 The Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we describe in detail the new requirements that we believe should be introduced to reservation-based systems and provide an outline of our solution. In Section 3, we introduce and describe the abstraction of Resource Controllers. In Section 4, we discuss the abstraction of Resource Control Lists and their interactions with resource controllers to make dynamic policy decisions. Our design and implementation of the various OS-level enhancements are discussed in Section 5. An empirical performance evaluation of our system is presented in Section 6. We finally present our concluding remarks and discuss future work in Section 7.

## 2 New Requirements for Reservation-Based Systems

In this section, we propose that resource usage be dynamically and flexibly controllable in reservation-based systems. We therefore introduce the following requirements.

### 2.1 Resource Access Control Capability

A basic question to consider when using reservations is, *"who is allowed to get what, how much, and for how long?"* For example, if we did not have any resource access control feature, the very first activity that entered the system would be able to get all the resources available. Admission control makes sure the system does not over-commit, but it does not control how much a certain class of activity is allowed to have. Essentially, it is a first-come-first-serve policy. Other requests that follow would not be able to get any resources in this scheme. This is often not an ideal behavior and can even defeat the entire purpose of reservations.

To prevent this problem, an administrator may want to grant reservations only to certain (trusted and privileged) activities. Also, he/she may want to restrict or control the *amount* granted to a certain class of applications at a certain *time*. Thus, one should be able to specify resource access control requirements and resource usage policy that is specified and enforced by the system.

### 2.2 Dynamically Defining Resource Principals

The recognition that it is often inadequate to charge resource usage to processes (in UNIX terminology), led to the concept of resource principals [1]. This made it possible to charge resources based on activities.

We argue that while the definition of a resource principal is important, a principal may not be fixed. That is, the definition of resource principal can change dynamically. For example, if we have a server serving multiple clients as illustrated in Figure 1, the `server` can be acting on behalf of distinct resource principals at different times. If the `server` is responding to `client_1`, it is working on behalf of `client_1` to achieve a common goal. Thus, it is natural to consider `client_1` and the `server` as a member of the same resource principal. The resource consumed by both the `server` and `client1` should be charged to the same entity. But, when the `server` is servicing `client_3`, it should be considered as part of a different resource principal. We do *not* want to charge the resource usage of the server to the resource principal associated with `client_1` in this

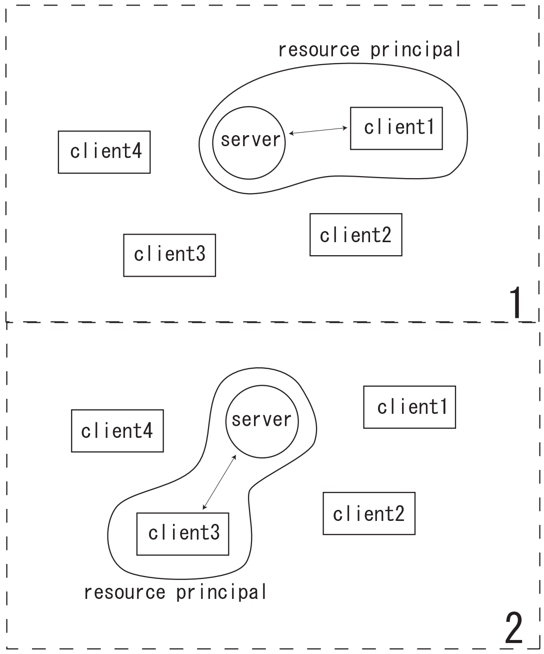


Figure 1: Resource principal for client server application

case.

In summary, there should be a way to specify such time-varying and context-sensitive changes in the definition of resource principals.

### 2.3 Detecting and Responding to Time-Varying Requirements

In previous reservation systems, the system was not able to capture the time-varying or dynamic behavior of activities. Rather, each application was responsible for expressing time-varying requirements by dynamically changing reservation parameters. Hence, the aggregation of these reservations used by applications may become inadequate when viewed as a whole from the systems perspective.[1] This is because the applications do not know the state of the system or their relationships with other programs (such as relative importance) that can change dynamically.

For the above reason, dynamic control over reservation should be governed by the system, with applications allowed to augment this decision by providing and/or exploiting application-specific knowledge. In other

[1] In many reservation-based systems, this problem was addressed by requiring applications to go through a central QoS manager which in turn made global decisions. But this required cooperation from all applications and typically required applications to be modified.

words, we propose that control over reservations be separated from the application and that the system be given more direct and fine-grained control. This separation makes it possible to detect and react not only to global system changes but also to changes in application state (such as the entry or exit of an application).

### 2.4 Separation of Policy from Mechanism

In many first-generation reservation-based systems, applications needed to be modified to benefit from reservations. The application programmer specifies the resource binding and reservation attributes explicitly in the program using the specification scheme provided. This is certainly a necessary feature for making end-to-end arguments [19]. We agree that applications should have control over resources, but *only when* this control does not collide with the system-wide policy. Ultimately, the system-wide policy should have precedence over specific requirements from each application. We treat application-specific demands as supplemental information.

Also, from a security perspective, letting the applications make all resource access decisions freely is often not desirable. For instance, to prevent DoS (Denial of Service) attacks on systems, the resource usage containment policy should be applied and enforced *before* giving applications any control over reservation. Even trusted applications can be used for instigating DoS attacks by improperly increasing their load or resource usage.

We finally argue that it is convenient and practical to make the resource usage policy specification be independent from any application or the operating system[2]. As a beneficial and intended side-effect, this enables legacy applications that do not embed any reservation requests to be incorporated into the reservation scheme *without* the need to access or modify any source code.

### 2.5 Terminology and Definitions

In order to address the above requirements, we introduce and/or clarify several abstractions. We first present the terminology and definitions used in the rest of this paper.

- *Activity*: An activity denotes a unit of computation for which the application or the system wishes to perform separate resource allocation and accounting; for example, the processing associated with a

[2] As noted above, the application could still embed specific requirements within the bounds of the system-wide policy specification.

single HTTP request is considered a single activity. An activity can span multiple protection domains.

- *Worker*: A worker is an active entity that performs a part of or the whole computation for an activity. In UNIX terminology, a worker can be a process or the kernel. For example, to process a single HTTP request, httpd and the CGI process, and the kernel responsible for network protocol processing are considered *workers* for this particular *activity*. A worker may be used to perform one or multiple activities.

- *Resource Controller*: A resource controller is a special class of worker whose primary job is to manage, allocate, control and track resources. For example, an operating system is a resource controller.

- *Resource Principal*: A resource principal is an entity to which resource usage can be charged. Thus, each *activity* should have a *resource principal*.

- *Event*: An event is an occurrence at a resource controller. It typically represents information about resources and their usage by a resource principal. This information may be asynchronously sent from one resource controller to another, and may also be synchronously queried.

### 2.6 Our Solution

Our system architecture consists of the following core interacting aspects:

First, resource controllers are used to manage, control and track resources at different levels of granularity. Resource controllers can be stacked on one another[3] and are discussed in detail in the next section.

Secondly, Resource Control Lists (RCLs) are specifications used by resource controllers and describe "*who* (resource principal) is allowed to get *what* and *how much*". Effectively, RCLs describe the policy which the resource controllers abide by. RCLs can change over time, and can therefore be used to capture the changing requirements on resources. For example, this enables one to solve the last part of the question, "who is allowed to get what, how much, and for *how long*?" Activities are granted resources as long as RCLs permit them to be. RCLs are discussed in detail in Section 4.

[3] A resource controller can also contain one or more resource controllers. For example, the OS resource controller can contain different resource controllers for network sockets and network interface cards. This aspect, however, is beyond the scope of this paper.

Finally, the definition of resource principals can change over time as the roles of workers performing computations change dynamically. We capture this by using events. A certain event, such as communication between a particular server and a client indicates that during this rendezvous, both the server and the client represent the same activity. Hence, it should be considered part of a single resource principal. We capture this dynamic collaboration between workers by *propagating* resource reservations between them. These dynamic interactions will also be discussed in Section 4.

## 3 Resource Controllers

In this section, we describe the primary functions of resource controllers and how they interact with one another. A resource controller's function is to

1 manage, allocate and control system resources,

2 observe and provide information about its own state.

There can be multiple resource controllers in the same system. But at the lowest level, we assume a resource controller that consists of a standard reservation-based operating system as described in [14, 1]. In this paper, we shall focus upon Linux/RK as the lowest-level resource controller which is based on the Resource Kernel [17] approach. However, the concepts and abstractions we present should also apply equally well to other related efforts.

One aspect we emphasize within a resource controller is its ability to support its own self-representation, analogous to the notion of a *reflective system* [12][4]. By self-representation, we mean that the internal state of the resource controller with regards to various resources is expressed explicitly.

It must be noted that a resource controller does *not* embed policy for resource control within itself. Each resource controller merely consults the policy description provided, and the control policy for resources controlled by a resource controller is provided through Resource Control Lists (RCLs).

[4] A reflective system is a system which incorporates structures representing aspects of itself. These structures are called self-representation of the system. Self-representation is causally connected to aspects of the system it represents. Computations on the self-representing data structure can actually bring modifications to itself.

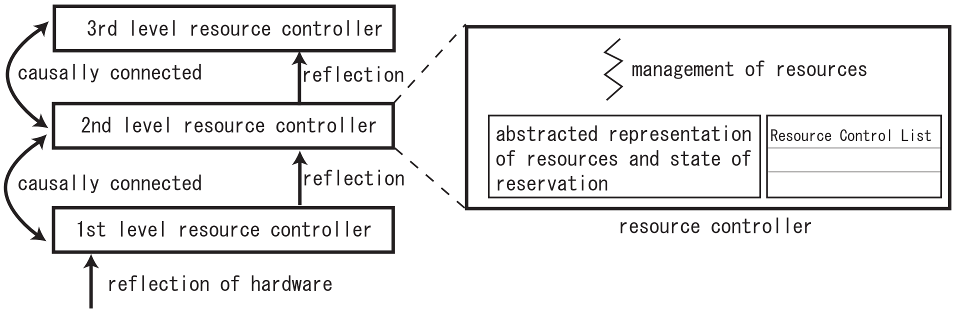


Figure 2: Resource controllers and self representation

### 3.1 Self-Representation in Resource Controllers

Resource controllers at different levels provide "self-representation of the system" at different abstractions with regards to resources. The self-representation provided by a resource controller allows it to answer questions about itself and to take actions on itself. This is illustrated in the architecture diagram of Figure 2.

The lowest-level resource controller corresponds to the OS (in our case, Linux/RK) and represents the state of the resources at the lowest abstraction as it provides the virtual representation of the actual hardware state. It also provides the state of different reservations within the OS[5] such as

- start time for each reservation
- period $T$ and computation time $C$ for each reservation
- % of resource used during the current period for each reservation

The time-scale used at this lowest OS level is typically in the range of tens to hundreds of milliseconds.

Above the operating system is the second-level resource controller, a "reflection" of the first-level resource controller. There can be various approaches to provide reflection of the lower-level controller. In our system, we generally consider a time-based abstraction, but other abstractions can also co-exist (as we shall discuss later).

The second-level resource controller provides state information of reservations at a higher level of abstraction.

[5] For more details on Linux/RK and its internal state, readers are asked to refer to [16].

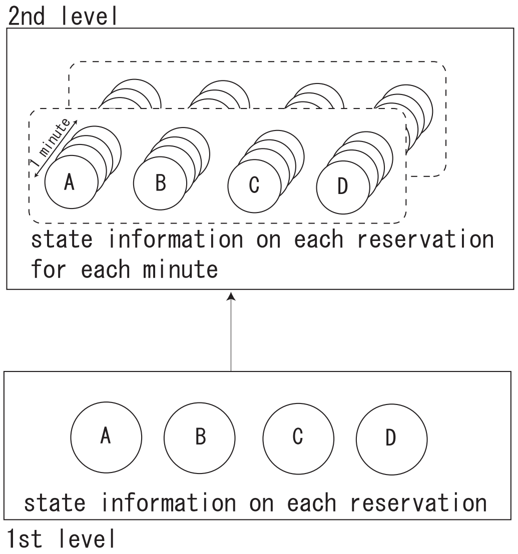

Figure 3: The time-based abstraction used in second-level resource controllers

The information exported is based on a longer (course-grained) time-span, whereas the first level provided information only on its *current* fine-grained state. We show below the state exported by the resource controller using the time-based abstraction with a granularity of minutes.

| |
|---|
| • average of used % of resource for each *minute* for each reservation |
| • average of reserved % of resource for each *minute* for each reservation |
| • maximum reservation used in each *minute* for each reservation |
| • minimum reservation used in each *minute* for each reservation |

Figure 3 illustrates how the information provided at the first level is summarized to provide state information at the second level. Then, there can be the third-level resource controller representing the behavior based on the granularity of *hours*.

### 3.1.1 Non-Time-Based Abstractions for Resource Controllers

As indicated earlier, abstractions other than time-based abstractions are also possible within resource controllers. For example, one can have a resource controller providing a self-representation of the first-level resource controller based on groupings of reservations. That is, several reservations which have the same

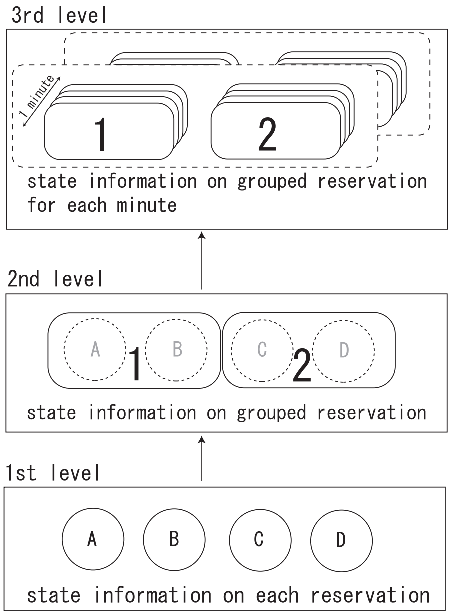


Figure 4: Inserting group based abstraction in second level

attribute can be grouped and treated as one entity. The state provided can be:

| |
|---|
| • reserved % of resource used this period for reservation related to the *same department* |
| • reserved % of resource reserved this period for reservation related to the *same department* |
| • maximum reservation used in this period for reservation related to the *same department* |
| • minimum reservation used in this period for reservation related to the *same department* |

Let us consider the example shown in Figure 4. In this example, the time-based resource controller used previously is placed as the third level to reflect the group-based second-level controller. In other words, controllers can be mixed and matched based on system requirements. In this case, the state visible at the resource controller with the time-based abstraction at the third level would be:

| |
|---|
| • average of used % of resource for each minute for reservation related to the same department |
| • average of reserved % of resource for each minute for reservation related to the same department |
| • maximum reservation used for each minute for reservation related to the same department |
| • minimum reservation used for each minute for reservation related to the same department |

This feature enables applications to query the state of reservations at different levels of abstractions. Also, in combination with RCLs, it lets resource controllers to manage resources using appropriate and flexible abstractions.

In the rest of the paper, we will focus on the resource control policies representable by Resource Control Lists and how they interact at different levels.

## 4 Resource Control Lists (RCLs)

In this section, we shall describe *Resource Control Lists* (RCLs) which are used to specify the control policies used by resource controllers. In a nutshell, one can control access to resource reservation in our system using RCLs at various levels of abstraction, and such policies can be changed dynamically. For instance, the behavior of applications can be tracked and enforced strictly by containing/punishing a malicious or uncooperative application. It is also possible to configure the system to satisfy administrative requirements such as giving more resources to bookkeeping, virus-finding and intrusion-detection activities at night-time or holidays.

### 4.1 The Basic Concept: Access Control

The basic idea behind RCLs has its origins in modern OS concepts. In most systems today, various system resources, most notably files, are managed through some form of access control. We argue that access to time-multiplexed resources such as CPU time or network bandwidth can, and should be, controlled in a similar fashion. Thus, when a reservation is about to be granted to a worker in an activity, its privilege is checked, just as it would be when opening a file. A subtle but core difference is that access to time-multiplexed resources may not be an "all or nothing" decision, as in the case of opening files. An activity may be granted partial access to the available time-multiplexed resource. Also, the amount of access allowed may change more dynamically and frequently compared to resources such as files.

RCLs will be specified in a configuration file by the system administrator or the program that has privileges to modify the RCL configuration file. Complex or optimal RCL configuration files can be generated automatically using tools.

### 4.2 Interaction Between Resource Controllers for Dynamic Control Using RCLs

We now describe how dynamic control over reservations is established through the interaction between multiple resource controllers. There are multiple RCLs in the system. They are used at multiple *levels* of the decision-making process at different abstractions. Generally, there are multiple RCLs in one resource controller. But only one RCL can be active at any given time in the same resource controller.

As stated already, resource controllers can be considered to be the active entities that implement the mechanisms to deploy the policies described in RCLs. The decision made by any resource controller can affect the behavior of the whole system since all controllers are causally connected ultimately to the underlying physical resources.

Resource controllers can compute (take action) based on their own abstraction levels. Specific actions possible at each resource controller depends on the abstraction it presents, and the interface between the lower level resource controllers. Rather than describing the specific actions a particular resource controller supports at each level, we will focus on how the control policy decision is propagated through multiple levels using default interfaces common to *all* levels of resource controllers.

There are two default interfaces that resource controllers must implement. These are: (a) sending and receiving events, and (b) setting RCLs of a lower level. The format of RCL at each level may vary but, the basic structure we take for RCL configuration files is to specify an event, and the action or access allowed for that event.

| | |
|---|---|
| **event** | **action taken / access allowed** |

The resource controller at each level uses the RCL in its own level to decide which RCL is to be deployed at the *lower* level. For example, the resource controller at the second level decides which RCL is to be used at the first level. Thus, the decision propagates from higher levels to lower levels.

On the other hand, events propagate from lower to higher level or to the same level. Events are used to trigger the decision-making process. At the lowest level, events are sent from the kernel (= first level resource controller) to the second-level resource controller. The events in the first level are pre-defined, and includes events indicating creation and termination of processes or an event indicating a reservation being depleted. (For example, we represent this event as **event⟨RPID: depleted⟩** where RPID is the resource principal ID.)

### 4.2.1 Example RCLs

In Figure 5, 3 levels of resource controllers are illustrated. At the lowest level (which is the kernel), there are 4 RCLs, `RCL[1]-1` to `RCL[1]-4`. The content of `RCL[1]-1` says resource principal $A$ is granted access to 10% of the CPU[6]. $B$ is granted 20%.

At the second level, there are 3 RCLs, `RCL[2]-1`, `RCL[2]-2`, and `RCL[2]-3`. The content of `RCL[2]-2` is illustrated in the figure. The first column in the table specifies the event this level is waiting on. The second column *"times/min"* describes the threshold for the event. If the event occurs more frequently than the specified threshold, the *action* specified in the third column is taken. Hence, the first entry in the table says, if **event⟨A: depletion⟩** is received once in 1 minute, take the action "set `RCL[1]-1`". Since resource controllers are only allowed to set the RCL of a level lower than itself, it will set the RCL of the first level. The second entry says, if **event⟨B:depletion⟩** is received 30 times in 10 minutes, take action "send **event⟨X⟩**".

Events can also be generated at the same level. There is a *clock* event in the third level. Clock events are generated by the resource controller itself and received at the same level. For example, **event⟨clock:24⟩** at the third level represents hours, and will be generated at the 24th hour, i.e. once everyday.

### 4.2.2 Interactions between Levels

We now describe how each level interacts with each other. Imagine RCL[1]-4 is the active RCL at the lowest level. If an activity represented by resource principal id $A$ uses up its reservation, an **event⟨A: depletion⟩** is sent by the kernel (= the first-level resource controller) up to the second-level resource controller. This triggers the second-level resource controller to set `RCL[1]-1` as the active RCL for the first-level resource controller, because `RCL[2]-2` at the second level is waiting for **event⟨A: depletion⟩** to occur once.

If the second-level resource controller was triggered to send **event⟨X⟩**, the third-level resource controller will

[6]Note that we have abbreviated the reservation information to just CPU % due to space limitation in many figures. In reality, computation time, period and deadline ($\{C, T, D\}$) are actually specified.

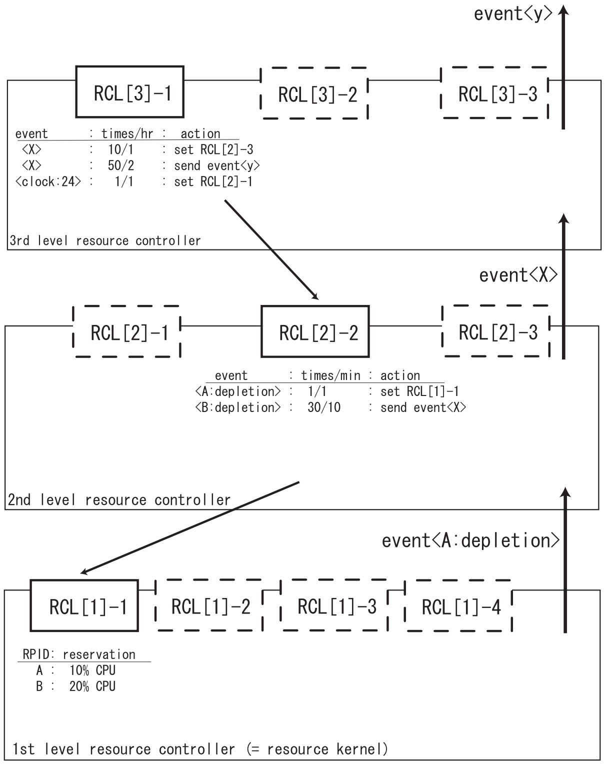


Figure 5: Resource Control List

receive this event. At the third level, `RCL[3]-1` is active. `RCL[3]-1` says if **event⟨X⟩** is received 10 times in 1 hour, then `RCL[2]-3` will be set as the active RCL for the lower level. If **event⟨X⟩** was received 50 times in 2 hours, then the third-level controller will send **event⟨y⟩** to the fourth level.

## 5 The Design and Implementation of OS-Level Enhancements

We have so far described the interactions among multiple levels of resource controllers and RCLs to change the system policy, based on default interfaces to send and receive events and to set RCLs. In this section, we describe additional features implemented at the operating system level (the first-level controller) specific to resource control. We first illustrate the RCLs used in the operating system level, and then describe the event-generating mechanisms used. We also describe the parental control feature that allows a parent to control the resources consumed by all its progeny.

In Figure 6, a detailed example of lowest level RCL is shown. Although the RCL file follows the generic format of describing an event and an action to take, we have separated it into two parts for illustration purposes

| RCL table for resource principal identification | | | |
|---|---|---|---|
| **discovery event** | **parental control** | **default** | **max** |
| RPID-X | YES | CPU=10% (C=50ms, T=500ms) | 10% |
| RPID-Y | NO | CPU=20% (C=20ms, T=100ms) | 30% |
| RPID-Z | YES | CPU=20% (C=40ms, T=200ms) | 40% |

| RCL table for IPC communication | |
|---|---|
| **IPC event** | **action** |
| IPC:RPID-X,RPID-S | PROPAGATE |
| IPC:RPID-Y,RPID-S | PROPAGATE |
| IPC:RPID-Z,RPID-S | DENY |

Figure 6: Content of the lowest-level RCL

since there are detailed attributes for each action.

The first part of Figure 6 shows the event indicating a discovery of resource principal and attributes for reservation that can be granted to that resource principal. The second part describes the action taken when there is communication between the principals.

### 5.1 RCL to Handle Discovery of Resource Principal

The first part of Figure 6 deals with events that are generated when resource principals are identified. As defined in section 2.6, resource principals are the entities representing a single activity. Hence, resource principals appear and disappear dynamically as workers for an activity enter and exit the system. An event (including RPID) is generated if a worker considered part of a resource principal enters or exits. The action field describes the reservation parameters to be used for that worker. If the parental control attribute is set to `yes`, all subsequent children forked by processes attached to this reservation will inherit this reservation. If the field is not set, a child of this process may be attached to another reservation if there is a matching entry in RCL. This feature allows an activity to ensure that the resources consumed by all its forked children are bounded by the parent's reservation. Remote entry points into a system (such as telnet, http, ftp and dial-ins) can use this feature to limit and track the resource usage for possible intrusions. Details on how and where the events are generated are described in Section 5.3.

According to the RCL listed in Figure 6, any worker associated with resource principal `ID=RPID-X` is granted access to resource reservation of 10% default and 10% max. If it is the first worker associated with `RPID-X`, it will create the reservation, and if not, the worker will be attached to the existing reservation. Workers associated with `RPID-Y` is granted access to 20% default and 30% max. Here, the definition of *default* and *max* is the following,

- *default:* This is the amount granted if the application does not explicitly request a reservation (such as legacy applications). The administrator/user of the machine can specify an amount he/she thinks is appropriate for that application. Hence, programs that are not tailored for Linux/RK would be given a 'default' amount. A specific application can request a different amount, but will be enforced to be less than or equal to the *max* value described next.
- *max:* This is the maximum amount a resource principal is allowed to have. The administrator's ability to specify the maximum is mainly to enable security or for containment reasons.

### 5.2 RCL for Communication Events

The second part of RCL in Figure 6 describes the action taken when communication events are detected. Communication events are generated when two workers communicate using some form of IPC. These events are mainly used to dynamically define the resource principal. Currently there are two possible actions taken in reaction to a communication event, PROPAGATE and DENY.

When PROPAGATE is specified, the reservation of the sender is propagated to the receiver. This is done by the resource controller transparently when the event is detected. For example, a shared server may want to inherit reservation from its clients (propagation). When a client requests a server for service (through a communication channel such as IPC), the server works on behalf of that specific client. But when a server is serving another client, it is working on behalf of a different client. Thus, the reservation used by that server should reflect the client it is serving at that time. Hence, clients will propagate its reservation dynamically as it makes its requests. The action PROPAGATE enables the dynamic changes of reservation based on communication requests.

If DENY is specified, the communication activity is denied. (The exact behavior of DENY is context-dependent, but might be, for example, a packet being dropped.) This is intended to facilitate prevention of

DoS attacks, or punishing greedy or malicious activities[7].

The lower part of the table shown in Figure 6 says that if a worker associated with RPID-X or RPID-Y sends a message to a worker associated with `RPID-S` (eg. shared server), PROPAGATE the reservation of the client. But any request from a worker associated with `RPID-Z` is DENYed.

Next, we describe how these events are generated.

## 5.3 Event-Generation Mechanisms at the OS Level

Several event-generation modules are appropriately placed in the system. There can be multiple modules in each resource controller. They generate events that resource controllers are interested in.

Here, we explain some examples of the event-generation modules placed in Linux/RK. All the event-generation modules are implemented as loadable kernel modules in our system.

### 5.3.1 System Call Monitor

The "system call monitor" registers filter functions into the system call table. These filter functions replace original system calls and are used to keep track of the activities in the system to identify the resource principal. When identified, it generates an event. Calls such as fork, vfork, clone, execve and exit are intercepted. Every time a particular system call is invoked from the user program, it will,

1. intercept the system call,
2. record necessary bookkeeping information,
3. forward control to the original system call,
4. after the original system call returns, updates book-keeping information, and
5. returns to the calling user as though the original system call has returned.

[7]Note that RCL files can change dynamically as described previously. Hence, one can easily implement a policy that DENYs for only a certain period of time.

| uid | gid | program name | once | RPID |
|---|---|---|---|---|
| 101 | * | java | yes | RPID-A |
| 101 | * | java | no | RPID-B |
| 101 | * | Doom | no | RPID-C |
| 101 | * | * | no | RPID-D |
| * | * | httpd | yes | RPID-E |
| * | * | cgi process | yes | RPID-E |
| * | * | * | no | RPID-F |

Figure 7: The `/etc/rk/syscallfilter/config` file

For example, when a fork system call is issued, it records the pid of the parent and child process and other information such as uid and gid. Subsequently, when that child process calls execve to load a new program image, it checks to see if that activity consists of a new resource principal. If it does, an event is generated. The resource controller will respond to that event if an action is specified in RCL.

The configuration file for system call monitor module consists of a table describing attributes for workers (often times "processes" in UNIX), and the matching RPID. An example is shown in Figure 7.

The ordering is important in this table. The system call monitor module will search from top to bottom, and adopt the first entry that matches and throws an event. If the `once` attribute is set to `yes`, it will be used by only one worker. Even if a second worker matches the attributes, the filter module will continue to look for the next match. If it is set to `no`, then it is used by multiple workers. Using the configuration above, if a user with uid 101 executes the following sequence, the resource principal id would be as shown below:

| **execution sequence** | **assigned** RPID |
|---|---|
| % `java` | `RPID-A` |
| % `java` | `RPID-B` |
| % `java` | `RPID-B` |
| % `Doom` | `RPID-C` |
| % `Doom` | `RPID-C` |
| % `netscape` | `RPID-D` |
| % `telnet` | `RPID-D` |

If a user with id other than 101 executes a httpd program, the RPID will be RPID-E and RPID-F otherwise. If httpd program spawns off a CGI process, it will be associated with the same resource principal as httpd. Hence, our system can treat the activity of 'serving a single HTTP request' as a single entity with regards to resource management[8].

Thus, resource principals can be defined in a flexible way so as to span multiple workers.

#### 5.3.2 Socket Filter

Another event-generation module placed in the OS layer is the socket filter module. It tracks the communication between workers. It observes the connection established, and monitors who is sending to whom by registering a filtering function. If there is an event that the resource controller is interested in, it generates an event. These events are used by RCL configuration files to PROPAGATE reservations. Currently the module keeps track of BSD-style interprocess communication.

## 6 Evaluation

In this section, we report the evaluation results of our infrastructure implemented on Linux/RK. All measurements were performed on a Pentium Pro 200Mhz machine with 64MB of memory.

### 6.1 Basic Cost of System Call Filter

Figure 8 shows the overhead incurred by the use of the system call filter module. It adds a constant of 30 to 50 *us* to the particular system call we monitor. Considering that system calls that we intercept such as fork, vfork, clone, execve, and exit are used relatively infrequently, we believe the overhead is of little concern to the overall performance.

### 6.2 Basic Cost of Socket Filter

Next, we show the cost of the socket filter module in Figure 9. Figure 9 shows the average cost of round-trip IPC for sending and receiving 1 byte data one million times. The first column (34.82 *us*) shows the cost

[8] Having parental control on for RPID-E in the RCL file has the same effect.

| system call | without filter | with filter |
|---|---|---|
| fork | 132.75 *us* | 160.35 *us* |
| exec | 3650.40 *us* | 3697.15 *us* |

Figure 8: Basic average cost of system calls w/ and w/o system call filters

| no filter | filter (no propagation) | filter (propagation) |
|---|---|---|
| 34.82 *us* | 37.20 *us* | 41.35 *us* |

Figure 9: Basic average cost of round-trip IPC w/ and w/o socket filters

without the filter which is the cost of IPC on Linux. The second column (37.20 *us*) shows the cost with our socket filter module loaded. Here, the socket filter module only monitors the communication. The third column (41.35 *us*) shows the cost when a reservation is propagated from the client to the server. Thus, all IPC between processes suffer an 2.38 *us* overhead even without reservation propagation. We believe that, this is acceptable considering the additional functionality provided. Also, we think that the impact of our filter in practical use is small due to the fact that it is part of a system call an application calls regardless of our module. Our filter modules will most likely suffer a cache miss only when the original system call does. We therefore expect that no significant disturbance to memory cache or overhead of context switch is being added.

### 6.3 Maintaining QoS using RCL

In this section, we demonstrate the ability of our system

- to protect important activities and contain the effect of unwelcome activities,
- to assign reserves to an activity spanning multiple workers,
- to service workers (legacy applications) transparently without modification to source code.

We performed a simple experiment illustrating how we can use RCLs to contain the effect of unwelcome activities in order to maintain QoS for a certain activity. In this experiment, we used mpeg_play that runs on Linux, but *without* making any changes to `mpeg_play` or any other programs used. In each case, we measured the CPU usage of every program at intervals of 200 *ms*.

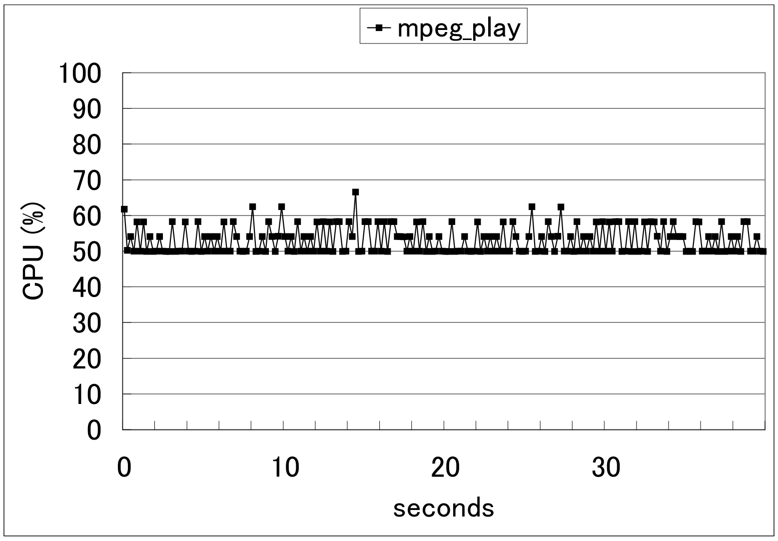


Figure 10: Running `mpeg_play` without competing activity, without reservation

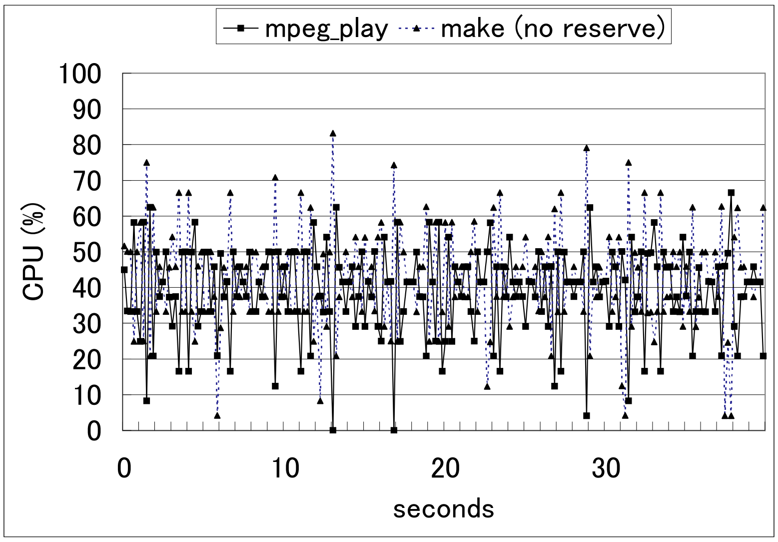


Figure 11: Running `mpeg_play` with competing activity, without reservation

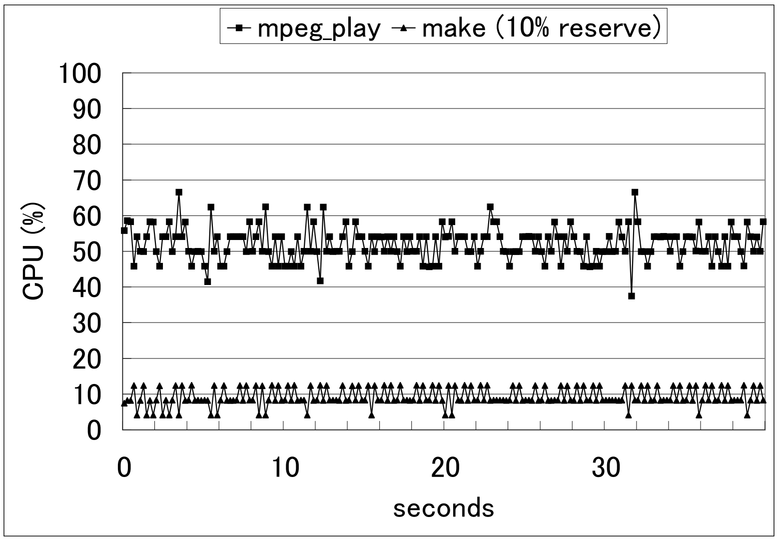


Figure 12: Running `mpeg_play` without reservation with competing activity attached to reservation

Figure 10 shows the CPU consumption of `mpeg_play` over time when no other competing workers are running. We see that it shows a fairly stable characteristic when there is no competition. As long as there are enough resources available, `mpeg_play` can provide adequate quality to the user.

Next, we introduce an activity that will compete against `mpeg_play`. The Linux kernel source code is compiled while `mpeg_play` is running. Figure 11 shows the case where resource reservation is not used. In measuring the CPU usage of kernel compilation, make, and *all* its child processes/workers are accounted for. This includes child processes such as gcc, cpp, ld, as, ar, sh, sed and cc1. As expected, `mpeg_play` is disrupted, the quality degrades and jittery behavior is seen.

Figure 12 shows the result when a reservation is used through a RCL specification. The system call filter parameters shown in Figure 13 and RCL illustrated in Figure 14 were used. The *parental control* flag field in RCL is set. All processes forked by make, including various compiler and linker programs such as gcc, cpp, ld, as, ar, sh, sed and cc1 are treated as part of the same activity and therefore are attached to the reservation specified by this RCL. The reservation created limits kernel compilation tasks to 10% ($C$=10ms, $T$=100ms) of total CPU time. It must be noted that `mpeg_play` is *not* attached to any reservations. The result shows

| uid | gid | program name | once | RPID |
|---|---|---|---|---|
| 101 | * | make | no | RPID-1 |

Figure 13: Configuration file for system call filter

| discovery event | parental control | default CPU | max |
|---|---|---|---|
| RPID-1 | YES | 10% (c=10ms, t=100ms) | 10% |

Figure 14: RCL for make

that the CPU usage of processes that are related to make is contained by the reservation and mpeg_play sees little effect. Even though we did not modify any part of *mpeg_play*, only a slight disruption from the make processes can be observed. The source of this disruption includes the effects from contention of disk I/O and page faults incurred by the virtual memory system.

We can say that mpeg_play shows the behavior close to the worst-case of an unreserved non-real-time application against maximum competition (with a large working set, and disk I/O such as make) that is reserved. To obtain better performance, we need to include additional features such as mapping of virtual memory to physical memory, and disk bandwidth reservation in our framework[9].

One may, of course, argue that the same result can be obtained if we were to reserve resources only for mpeg_play. That is true since we are using the same technology to achieve our results. The novelty in our approach lies in its complete flexibility. The administrator can use an appropriate RCL configuration to specify various targets using wild-cards and parental control properties. One can also define an activity that spans multiple workers and attach it to a resource reservation. Not only can one attach reservation to specific target(s), one can use it to contain the undesired effect of *other* activities *transparently*. This includes the effect from programs that are not known in advance.

### 6.4 Dynamically Defining the Resource Principals

Next, we show an experiment to illustrate our system's ability to define a resource principal *dynamically* using propagation of reservation and *transparently* without modifying the application.

In this experiment, we have one server that serves the client by computing for a certain amount of time and

[9] Disk and network bandwidth reservation have been implemented on our previous version of our resource kernel (RT-Mach). We are currently porting these features to Linux/RK. Wiring down of memory is supported in Linux using the mlockall and mlock system calls.

returning the result to the client. There are multiple identical clients. Only one client has a reservation. We programmed the server using *select()* system call such that it gives precedence to clients with reservations when there are multiple client requests pending. We have measured the response time of the server when serving the client with reservation. Results are presented in Figure 15.

First we measured the client's response time without reservation propagation. That is, when the client with reservation requests service from the server, the reservation is not propagated. Hence, the server will compute without reservation. If we consider the work by the server responding to the client and the work done by that particular client as one activity, only part of the work is charged to the reservation. In this case, as we increase the number of competing (identical) clients, we observe that the response time of the server increases linearly. This is caused by the server competing with other clients for resource.

Next, we measured the response time with reservation propagation. A reservation is propagated to the server as soon as the client with reservation requests service. The reserved client and the server (only when serving that client) are considered members of a resource principal and workers of a single activity. Note that when the server is responding to another unreserved client, it is considered a part of the other resource principal. Thus, the resource principal used is modified dynamically.

As shown in Figure 15, the response time is almost equal to the case when there is no competition from other clients. The slight increase in response time is due to two reasons. One is the overhead of *select()* system call that increases linearly with the number of sockets, as reported by [1] and others. The second reason is because the request from the reserved client may block (for at most one client request) when the server is responding to others. The server will not preempt its service but rather complete its ongoing computation, and only then respond to the reserved client. But the server will always choose a reserved client (in priority order if multiple reserved clients are queued) if present. So, the reserved client is guaranteed to wait for no more than one service request.

We have therefore shown that without the ability to define a resource principal in a dynamic and flexible way, QoS requirements cannot be met.

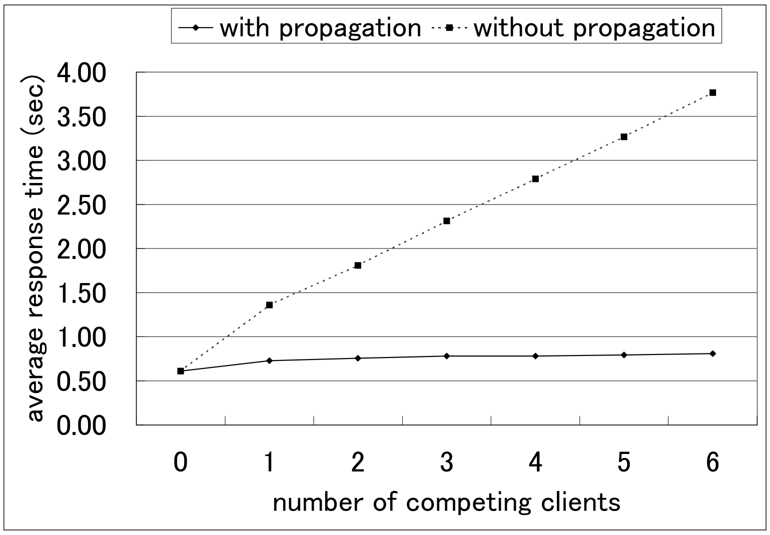


Figure 15: QoS provisioning by propagating reservation

### 6.5 Changing the Behavior of the System Using RCLs

Next, we show an application using multiple RCLs and multiple resource controllers. In this experiment, we demonstrate the use of multi-level resource controllers to

- detect changes in resource usage due to any potentially undesirable activity (intrusion detection),
- react to the changes in application behavior.

Here we assume the following scenario. We define resource principal `RPID-X` as any worker that cannot be trusted. Specifically, any program that is not executed with the path /usr/sbin/, /usr/bin/, and /usr/local/bin is categorized as a member of resource principal `RPID-X`. (This can be easily specified in the system call monitor configuration file using wild cards.)

We assign a CPU reservation of 30% to `RPID-X`. The second-level resource controller monitors the times `RPID-X` depletes the reservation. If `RPID-X` continues to deplete (uses up) its reservation for a certain periods of time, we recognize this as potential intrusion activity to the system, and restrict its resource consumption to 10% until we know for sure. At the same time, we trigger pseudo-intrusion detection software that usually has very low CPU resource reservation assigned to it by giving it 40%.

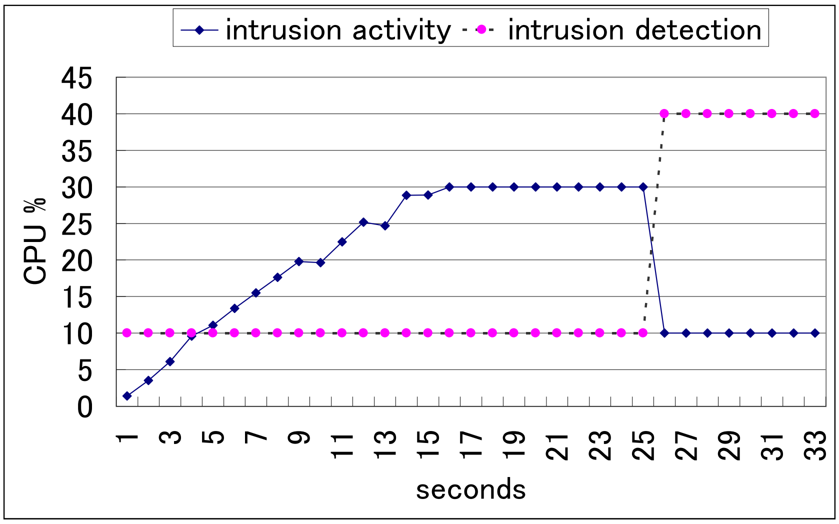


Figure 16: Detecting resource usage changes

Figure 16 shows the resource consumption of each activity. We increased the load for intrusion activity (`RPID-X`) gradually. As the figure illustrates, the second level resource controller reacts when the reservation is used up continuously and sets the reservation to 10%.

## 7 Conclusion and Future Work

In modern operating systems, various system resources, notably files, are managed through some form of access control. In this paper, we have argued that access to time-multiplexed resources such as CPU time or network bandwidth can, and should be, controlled in a similar fashion. Based on this simple but core concept, we have proposed and evaluated the use of a new facility called Resource Control Lists (RCLs). RCLs enable systems with resource reservation capability to control resource usage in a dynamic manner following a specified resource usage policy.

RCLs can also change over time. Thus, policies that change over time depending on the state of the system or the application can be flexibly described. Also, the definition of resource principals change over time as the role of workers changes dynamically. We capture this by using events and resource propagation. Further, a parental control feature extends our ability to recognize the dynamic behavior of applications by managing the resource usage of child workers. Our notion of stackable resource controllers also enables resource management and tracking at different levels of granularity. Our empirical results on Linux/RK showed that the overhead imposed by our abstractions is acceptable. We have also illustrated how these abstractions can be used by programs such as intrusion-detection utilities to track and react to events as they occur. We have also demonstrated that without our abstractions, QoS requirements cannot always be met.

We believe that the abstractions of stackable Resource Controllers, flexible and dynamic Resource Control Lists and propagation will allow operating systems to not only support QoS requirements but also help in building secure and easily maintainable systems. Our future work on this topic will include end-to-end support for resource controllers, ease of configuration, and administrative tools.

## References


[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[2] J. C. R. Bennett and H. Zhang. $WF^2Q$: Worst-case Fair-Weighted Fair Queueing. In *Proceedings of IEEE INFOCOMM'96*, Mar. 1996.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In *Proceedings of USENIX Annual Technical Conferance*, June 1999.

[4] C. Liu and J. Layland. Scheduling Alogritms for Multiprogramming in a Hard Real Time Environment. In *JACM*, volume 20(1), pages 46–61, 1973.

[5] M. Caccomo, G. Lipari, and G. Buttazzo. Sharing Resources with the TB* Server. Technical Report TR 04-99, Retis Lab, Scuola Superiore S. Anna, Apr. 1999.

[6] Ensim corporation. www.ensim.com, as of 12/99.

[7] S. Ghosh and R. Rajkumar. Network Bandwidth Reservation using the Rate-Monotonic Model. In *Proceedings of SoftComm'99*, Oct. 1999.

[8] I. Leslie and D. McAuley and R. Black and T. Roscoe and P. Barham and D. Evers and R. Fairbairns and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. In *IEEE Journal on Selected Areas in Communication*, 1995.

[9] IBM. *AIX V4.3.3 Workload Manager Technical Reference*. Feb. 2000.

[10] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1998.

[11] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 93.

[12] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference*, Dec. 1987.

[13] M.B.Jones, D. Rosu, and M-C.Rosu. CPU Reservation and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 1997.

[14] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[15] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of the 18th IEEE Real-time Systems Symposium*, Dec. 1997.

[16] S. Oikawa and R. Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proceedings of the 1999 IEEE Real-Time Technology and Applications Symposium*, June 1999.

[17] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-time and Multimedia Operating Systems. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, Jan. 1998.

[18] S. Saewong and R. Rajkumar. Cooperative Scheduling of Multiple Resources. In *Proceedings of the 20th IEEE Real-time Systems Symposium*, Dec. 1999.

[19] J. Saltzer, D. Reed, and D. Clark. End-to-End Arguments in System Design. *ACM Transaction on Computer Systems (TOCS)*, 2(4), Nov. 1984.

[20] J. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Hard Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

[21] SunMicrosystems. *Solaris Resource Manager 1.0 White Paper*. 1998.

[22] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, Oct. 1990.

[23] Y.-C. Wang and K.-J. Lin. Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. In *Proceedings of IEEE Real-time Systems Symposium*, Dec. 1999.

[24] J.-S. L. Z. Deng. Scheduling Real-Time Applications in an Open Environment. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1997.