Protecting Resources with Resource Control Lists

Akihiko Miyoshi and Ragunathan (Raj) Rajkumar
Real-time and Multimedia Systems Laboratory
Departments of Electrical & Computer Engineering and of Computer Science
Carnegie Mellon University
miyos@cmu.edu, raj@ece.cmu.edu

Abstract

Resource management has become an important issue for computer systems as QoS-sensitive multimedia applications and hostile acts such as denial of service (DoS) attacks become widespread. Since applications access the processor, network and other subsystems, attacks on any of these subsystems can lead to undesirable behavior. To maintain system integrity even under such attacks, an unprivileged application must not be allowed to intentionally or unintentionally affect the progress of others. In this paper, we present a first-class abstraction called Resource Control Lists (RCLs) *to specify and enforce protection policies on time-multiplexed resources, similar to access control on files. RCLs put access control on time-multiplexed resources such as CPU time and network and disk bandwidth. RCLs are practical, flexible and provide several benefits. Protection policies specified by RCLs can be under administrator control, customized to a site, and can be dynamically adapted.*

We have built support for RCLs into Linux/RK, a real-time version of Linux that provides resource reservations where applications receive a promised amount of resources by making reservations. Quantitative measurements with limited forms of DoS attacks show that our new features provide flexible functionality while imposing acceptable overhead without modifying the applications. We also demonstrate that resources are protected even in the face of malicious activities. Finally, we show how RCLs can be used to deliver resource protection on multiple web server configurations including the hosting of multiple logical sites by an ISP and preferential treatment of buyers on e-commerce sites.

1 Introduction

Resource management is a primary task of an operating system such as UNIX or Windows NT. In such timesharing operating systems, resources are distributed so that the throughput of CPU intensive applications and the responsiveness of interactive applications are maximized. But in these systems, security and protection aspects of resource management are not considered. For example, if one greedy process consumes large quantities of a resource, it is very likely that other innocent activities will see a direct effect and suffer. If we run an application with timing requirements such as a movie player under very high load, we observe jitter and degradation of frame-rate.

As the demand for real-time and multimedia applications increases, various QoS provisioning infrastructure and mechanisms that extend basic resource management facilities have been introduced. One of the well-accepted techniques is *resource reservation*. Unlike resource management schemes on traditional time-sharing operating systems 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. Variants of this technique have been implemented not only on special-purpose systems, but also on many general-purpose systems such as RT-Mach[10], Linux[13], Windows NT and other UNIX-based systems[1, 2, 8, 18].

Another resource management issue that has become increasingly important is to make sure that the integrity or the health of the system is maintained. In the face of hostile acts such as denial of service (DoS) attacks, or increases in the use of mobile code with varying levels of trust, it is critical that resource usage is controlled and follows a pre-defined usage policy. The policy should address the security concerns of the system and be actively enforced. Especially in a reservation based system, this is critical.

If the first activity in the system obtains a reservation of 100%, any subsequent activities will not have any resources left to use. Hence, one must make sure that a particular application or group of applications only gets access to resources that it is allowed to have, and cannot intentionally or unintentionally prevent others from accessing their needed resources. Thus, resources must be protected. The system must make sure the reservation granted to a particular activity follows a defined policy rather than a first come first serve policy. We refer to this as policy-based resource protection.

We address these issues 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 represented by a reservation), and the *policies* for allocating resources based on application-level and system-wide requirements. With the use of RCLs, an administrator of the system obtains direct control over resources rather than requiring or letting application pro-

grams to embed the resource control policies.

2 New Requirements for Protection

One required new policies and mechanisms to *actively* enforce and *contain* resource usage of unwelcome and malicious activities. In this section, we discuss the new system requirements that arise.

2.1 Access Control on Resources

One goal of protection is to make sure that the appropriate amount of resources goes to legitimate users. For example, if reservation (or more general, resource itself) goes into the hand of a wrong user, there is no sense in using reservation. Thus, 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.

To prevent this problem, any request regarding allocation and assignment of resources should follow a certain *access control* policy. For example, an administrator may want to grant resources (in the form of reservations) to certain (trusted and privileged) activities to guarantee the progress of those activities, and at the same time, he/she may want to restrict or control the *amount* granted to a certain class (identified as hostile or unwelcome) of applications.

2.2 Policy Enforcement at Resource Demultiplexing Points

A culprit can cleverly bypass any intended policy enforcer by placing itself between the point where resources are demultiplexed and where the policy enforcement mechanisms are. There are several ways to cause resources to be demultiplexed and accessed. A typical route may be via system calls or (network) interrupts. One can create a new process which requires allocation of CPU time and memory space by using system calls. An interrupt generated by receiving packets will require resources to process and respond to it. Also, communication paths such as IPC can be used to gain access to resources *indirectly*. Unless all these access paths are monitored, there is the possibility of unrestricted access to resources which can be used to violate the system by bypassing the resource management subsystem. Even if all of the applications running are somehow made trustworthy, a centralized resource manager or a QoS server still cannot prevent inadequate access to resources from the network. One such phenomenon is known as the *receiver live-lock*[5] problem. To ensure that the protection policy is strictly enforced, loopholes or opportunities to exploit a covert channel should not exist. Thus, protection policies should be enforced where the resources are demultiplexed.

2.3 Active and Transparent Enforcement of Protection Policy

Many of the previous reservation based systems designed for QoS provisioning assume that applications possess negotiation capabilities with the central resource manager (eg. QoS manager) or the operating system. This implies: (1) all the applications know the protocol to communicate with the central server/OS and, (2) they will follow the decision of the server. However, any application can be potentially hostile and malicious. There may also be vast amounts of legacy applications that cannot be modified.

Applications may not explicitly request reservations (eg. legacy applications), or they may try to access resources indirectly (eg. through IPC with server programs or other covert channels). These activity should be identified and enforced with a protection policy automatically without cooperation from the application itself. Thus, the policy should be enforced *actively*, *transparently* and *forcefully*.

2.4 Resource Principals: Resource Charging Entity

In traditional operating systems, resources are accounted for and enforced based on processes. Originally, processes represented protection domains (with regards to virtual memory and address spaces). However, processes itself do not completely capture the way resources are consumed on behalf of activities. For example, multiple processes may collaborate to form a single activity. Or a single process may work on behalf of multiple activities.

The recognition that it was often inadequate to charge resource usage based on processes, led to the concept of resource principals[1] and resource sets[10, 14]. The accounting of resources used by multiple processes in different protection domains but cooperating to achieve a common goal, is facilitated by an abstraction called the *resource principal*. A resource principal is an accountable entity to which the system can charge resource usage for a certain activity. Here, 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 single `HTTP` request can be considered a single activity. An activity can span multiple address spaces. For example, at one point in time, if the HTTP server and a CGI process are collaborating to serve a request from a particular client, we would say that these two processes are associated with a single activity and treat them as a single entity.

We propose to account and enforce resources based on resource principals. Since we do not assume any cooperation from the participating applications, the identity of the resource principals should be constructed *transparently* to the application. For example, we do not assume that an application to declare itself associated with a particular resource principal, or even attempt to explicitly join a resource principal.

2.4.1 Dynamically Defining Resource Principals

While the definition of a resource principal is important, a principal may not be fixed for any activity. In other

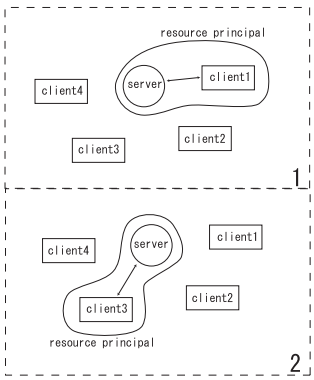

Figure 1. Resource principal for client server application

words, the binding to resource principal can change dynamically. Thus, one should be able to capture this dynamic nature. 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`. 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 case because they may have different protection requirements.

In summary, there should be a way to specify such time-varying and context-sensitive changes in the binding to resource principals and do so in a transparent manner without the need to modify the applications.

3 Our Solution: RCLs

In this section, we propose our solution to provide protection facilities that meets the requirements.

First, Resource Control Lists (RCLs) are specifications that describe "*who* (resource principal) is allowed to get *what* and *how much*". Resource Control Lists are specified at resource demultiplexing points in the operating system such as system call entry points. Effectively, RCLs describe the policy which the resource demultiplexers abide by. This enables the system to enforce protection policies exactly where the resources are demultiplexed and implement a system-wide policy. 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.

Further, binding to resource principals can change over time as the roles of processes performing computations change dynamically. We capture this by using `events`. An event is an occurrence in the system that can be captured and subsequent response can be controlled by a RCL. 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 processes by *propagating* resource reservations between them. We thus construct the identity of resource principals through observing the activity of applications. The administrator will describe in a configuration file, the attributes of the resource principals the system is to identify. If the system were to identify a resource principal that serves a particular client requesting for a dynamic web page, it would observe the following activities: 1) receipt of packet from client source address, 2) the HTTP server receives the request, 3) if necessary, HTTP server forks a CGI process, 4) IPC between HTTP server and CGI process 5) send data back to client through network. All these operations are monitored by the system to transparently identify a new resource principal with HTTP and CGI process as a transient member.

3.1 Resource Control Lists: RCLs

The basic concept of RCL is based on an observation of modern operating systems. 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.

event	threshold	action taken / access allowed

Unlike access control lists for files, a RCL lists *events* and a *threshold* as the target rather than the object (resource principal) itself as shown above. This is because we assume that the definitions of resource principals can dynamically change. Associated with the events are the *actions* or *access allowed*. Example RCLs are presented presented in Figure 3. If the system observes any activity that results in the change of resource usage and the state of the resource principal, a synchronous event is generated and is used to make resource management decisions as shown in Figure 2. An explicit request for resources is an event. The detection or creation of a resource principal is an event. The changes in the definition of a resource principal is an event. More specifically, an event can be the creation of a process which may result in the detection of a new resource principal. It can be the reception of a network packet, which also may result in the detection of a new resource principal. It may be IPC between two processes that results in the two collaborating to achieve a common goal. In this case, one of the process may be temporarily included in a resource principal representing that activity.

When an event is generated and it's value is higher than the threshold value, one of two things happen based on the action/access allowed field of the RCL. A) If the event was the result of an explicit attempt to access a resource, its access privilege is checked and enforced. B) If the event was due to other causes, an action is taken. An action can be to:

1. Associate the (newly identified) resource principal to a specific resource reservation - Every process in the system will be associated with a resource principal. And

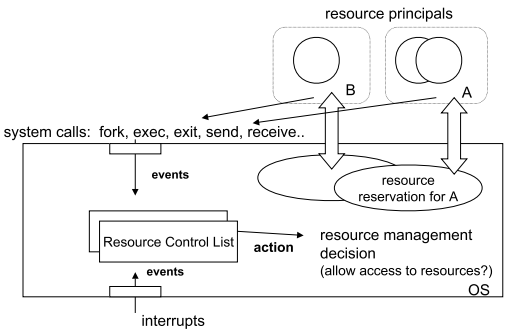

Figure 2. Architecture

every resource principal will be assigned a reservation. [1] By controlling the attributes of resource reservation, we control the resource usage of the resource principals.

2. Change the attributes of an existing resource reservation - for instance, if the state of the resource principal changes.
3. Change the entry of the RCL - In some sense, RCL can be considered to be a particular state instance in a state machine, which adapts as the state of the application and the system changes. A threshold value is used to control when to trigger the action, eg. if an event occurs more frequently then the threshold 10times/sec, then take action.

4 Implementation

We have implemented our facilities on Linux/RK, a real-time extension of Linux which provides resource reservation facilities and incorporates resource kernel (RK) features. Figure 2 illustrates our architecture. The RK feature allows guaranteed, timely access to CPU cycles [14], disk bandwidth [11] and network bandwidth by providing the resource reservation abstraction.

4.1 Resource Control Lists: RCLs

We now describe the details of the RCLs. Here, we also describe the parental control feature that allows a parent to control the resources consumed by all its progeny. There are multiple specifications of RCLs for various resource demultiplexing opportunities. In Figure 3, a detailed example of RCL is shown. Although each RCL file follows the generic format of describing an event and an action to take, they each have detailed attributes for each action.

The first part of Figure 3 deals with an event indicating a **discovery of resource principal** and attributes for reservation that can be granted or enforced to that resource principal. The second part describes the action taken when there is **communication** between the principals.

4.2 RCL to Handle Discovery of Resource Principal

The first part of Figure 3 deals with events that are generated when resource principals are identified. These resource principals appear and disappear dynamically as

[1] Note that in our implementation of the reservation, we allow the specification of reservation without guarantees which is equivalent to time-sharing. But this is different from 0% reservation.

RCL **table for resource principal identification**

discovery event	threshold	parental control	default	max
RPID-U	1/1	set new rcl entry		
RPID-V	1/1	YES	CPU=5%	10%
RPID-W	1/1	YES	NOGUARANTEE	10%
RPID-X	1/1	YES	CPU=10%	10%
RPID-Y	1/1	NO	CPU=15%	20%
RPID-Z	1/1	YES	CPU=20%	20%

RCL **table for IPC communication**

IPC event	threshold	action
IPC:RPID-X,RPID-S	1/1	PROPAGATE
IPC:RPID-Y,RPID-S	1/1	PROPAGATE
IPC:RPID-Z,RPID-S	1/1	DENY

Figure 3. Example of RCL specification

processes for an activity enter and exit the system. An event is generated if a process considered part of a resource principal enters or exits. The action field describes the reservation parameters assigned for that process. If the parental control attribute is set to `yes`, all subsequent children forked by a member of this resource principal will be included in the same resource principal, and hence will inherit this reservation. If the field is not set, a child of this process may be attached to another reservation if it is classified as another resource principal. 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.

According to the RCL listed in Figure 3, any process associated with resource principal `ID=RPID-X` is assigned resource reservation of 10% default and 10% max[2]. If it was the first process associated with `RPID-X`, the system creates the reservation, and if not, the process is attached to the existing reservation assigned to `RPID-X`. Processes associated with `RPID-Y` is granted access to 15% default and 20% max. `RPID-W` will be assigned a reservation with `NOGUARANTEE`, which is the equivalent to timesharing.

Default is the amount assigned (and forcefully enforced) if the application (such as a legacy application) does not explicitly request a reservation. Programs that were not tailored for Linux/RK would be given a 'default' amount by the administrator. An application tailored for Linux/RK can request a different amount, but will be enforced to be less than or equal to the *max* value. *Max* is the maximum amount a resource principal is allowed to have if it explicitly requests for the resource.

Finally, when the discovery event for RPID-U is generated, the RCL entry is modified. This is useful for expressing requirements such as "When a gold card member enters the system, modify the current RCL listing so that it will reduce the resource usage amount of other non-gold card members to a smaller value".

4.3 RCL for Communication Events

The second part of RCL in Figure 3 describes the action taken when communication events are detected. Communication events are generated when two entities communicate via IPC. Currently there are two possible actions taken in reaction to this event, PROPAGATE and DENY.

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

When PROPAGATE is specified, the reservation of the sender is propagated to the receiver. This is done by the system 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 IPC), the server works on behalf of that specific client. Thus, the reservation used by that server should reflect the client it is serving at that time. Hence, the reservation of the client is propagated as it makes its requests. Note that when the `lock` field in the system call monitor configuration is set (Figure 4), propagation is not permitted since the process is permanently attached to that original reservation. If DENY is specified, the communication activity is denied. (The exact behavior of DENY is context-dependent, for example, a packet being dropped.) This is intended to facilitate prevention of DoS attacks, or punishment of greedy or malicious activities. Note that RCL entries can change dynamically as described previously. One can implement a policy that DENYs for only a certain period of time.

The lower part of the table shown in Figure 3 says that if a process associated with `RPID-X` or `RPID-Y` sends a message to a process associated with `RPID-S` (eg. shared server), `PROPAGATE` the reservation of the client. But any request from a process associated with `RPID-Z` is DENYed.

4.4 Event-Generation Mechanisms

Event-generation modules are implemented as loadable kernel modules and are appropriately placed in the system.

4.4.1 System Call Monitor Module

The system call monitor module registers filter functions into the system call table. These filter functions intercept 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 bookkeeping information, and
5. returns to the calling user as though the original system call has returned.

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 described by the administrator. If it does, an event is generated. The system will respond to that event if an action is specified in the RCL.

The configuration file for system call monitor module consists of a table describing attributes of the activities, and the matching resource principal ID (RPID). An example configuration is shown in Figure 4. 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 indicating discovery of a resource principal. If the `once` attribute is set to `yes`, it will be used by only one process. Even if a second instance matches the attributes, the monitor module will continue to look for the next match. If it is set to `no`, then it is used by multiple processes. If the `lock` field is set, the process is permanently associated with the resource principal and to the reservation attached to it. A locked resource principal cannot receive reservation that is propagated.

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

Figure 4. The `/etc/rk/syscallmonitor/config` file

As an example, using the configuration above, if a user with uid 101 executes the following sequence, the resource principal ID assigned would be as shown below:

execution sequence	assigned RPID
% `java`	`RPID-A`
% `java`	`RPID-B`
% `java`	`RPID-B`
% `mpeg_play`	`RPID-C`
% `mpeg_play`	`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 without explicit cooperation from the participants.

4.4.2 Communication Monitor Module

Another event-generation module placed in the operating system is the communication monitor module. It has two responsibilities. The first is to track communication between resource principals. It observes the connection established, and monitors who is sending to whom by registering a filtering function. If there is an event that the system is interested in, it generates an event. Currently the module keeps track of BSD-style IPC.

The second responsibility is to identify the resource principals that may be created as a result of packet reception. For example, any activity associated with serving a HTTP request from client-A can be treated as a single resource principal. In this case, the module monitors the incoming packet from client-A going to the HTTP server. Currently IP based communication is monitored.

5 Evaluation

In this section, we report the evaluation results of our infrastructure implemented on Linux/RK based on Linux

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

Figure 5. Basic average cost of system calls w/ and w/o system call monitors

no filter	filter (no propagation)	filter (propagation)
34.82 us	37.20 us	41.35 us

Figure 6. Basic average cost of round-trip IPC w/ and w/o communication monitor

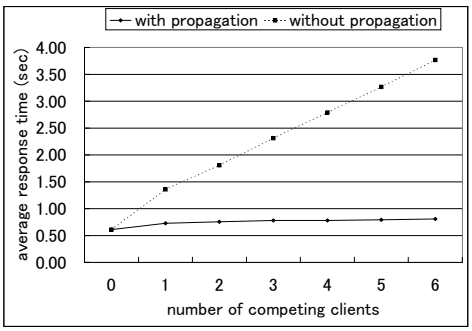

Figure 7. Impact of Reservation Propagation

2.2.14. All measurements were performed on a Pentium Pro 200Mhz machine with 64MB of memory.

5.1 Basic Cost of Monitor Modules

Figure 5 shows the overhead incurred by the use of the system call monitor module. The system calls that are affected by the modules are calls such as fork, vfork, clone, execve, and exit. The overhead is due to extra procedure calls into our functions and look-up access to data structures representing RCLs.

Figure 6 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 without the communication monitor module which is the cost of IPC on Linux. The second column (37.20 us) shows the cost with our module loaded. Here, the 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.

Our implementation of our filters is not optimized for performance yet, since our current focus is on functionality. In future versions of our implementation, scalability and performance of the modules will be considered.

5.2 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. We demonstrate that resource reservation is not effective without the ability to capture the dynamic behavior of applications.

In this experiment, we have one server that serves the client by computing for a certain amount of time and returning the result to the client. The server is initially identified as a resource principal with only timesharing access to resources. There are multiple identical clients. One client is identified as important and assigned a guaranteed amount of reservation. All the others will not, and will be considered unimportant. The *select()* system call used by the server gives precedence to the important client when there are multiple client requests pending. We have measured the response time of the server when serving the important client with reservation. Results are presented in Figure 7.

First we measured the client's response time without reservation propagation. The server will compute without reservation from the important client. 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 being disrupted by the other clients.

Next, we measured the response time with reservation propagation. The reservation is propagated to the server as soon as the important client with reservation requests service. The reserved client and the server (only when serving that client) is considered the member of a single resource principal. When the server is responding to another unreserved client, it is considered a part of another resource principal. Thus, the resource principal which the server belongs to is modified dynamically. As shown in Figure 7, 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 the *select()* system call that increases linearly with the number of sockets, as reported by [1] and others. The second reason is that 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 the highest priority reserved client 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, the use of resource reservation is not very effective. It also indicates that without reservation propagation, the resource usage of an important activity can easily be disrupted by other (potentially hostile) activities. Some of the theoretical implication on using propagation with regards to schedulability is discussed in [4].

5.3 Applying RCLs to a Web Server

In this section, we apply RCLs to provide various web server models using unmodified Apache httpd server.

Web hosting services are very common today. A server may host multiple logical web servers (eg. `www.A.com` and `www.B.com`) on the same physical machine. In these environments, it is important to contain and isolate the resource usage of one activity/site from the other. For example, even if `www.A.com` was a very busy site, ideally `www.B.com` should not be affected. It should always have access to resources promised. In this case, RCLs should

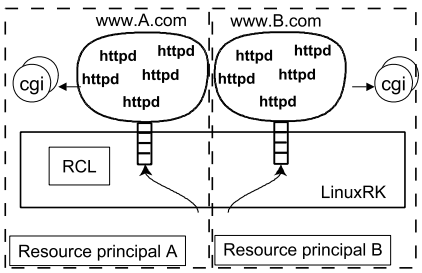

Figure 8. Separate web server on same machine

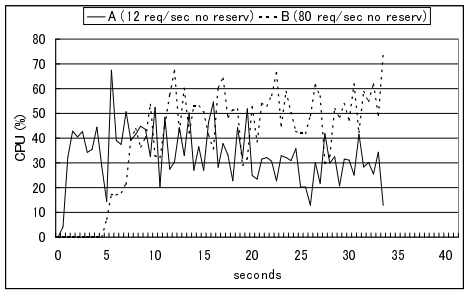

Figure 9. Separate Server, Different Rate (Resource principal A=low request rate, Resource Principal B=high request rate), *Without* RCLs

be configured so that both `www.A.com` and `www.B.com` are distinct resource principals and assigned a reservation. If `www.A.com` was granted 40% CPU and `www.B.com` with 20%, `www.A.com` should always get 40% of the resource no matter how popular and busy `www.B.com` is, and vice versa when `www.A.com` is busy or popular.

In another case, a single site may have several classes of co-existing clients. A client actually shopping at an e-commerce site can be treated as one class of client, and the others that are just browsing at the same site can be classified as another. Then the e-commerce site may want to promise the buying customer consistent service without being disrupted by the activities of the browsing customers. One can apply RCLs to define all customers shopping as one resource principal and the set of all customers browsing as another.

To test these services, we evaluated three different service models as shown in Figure 8,11,and 14 using the Apache web server. All configuration and definition of resource principals were specified through RCLs without modifying any of the programs used.

5.4 Separate Web Server on Same Machine

The first model illustrated in Figure 8 has separate instantiations of the web server. They wait on different ports. To host multiple logical web sites, each instantiation of the server is associated with one logical web site. Note that Apache spawns off multiple `httpd` daemons for a single server. We created two web servers (server A and server B), each as a separate resource principal. Through the specification of RCLs, resource principal associated

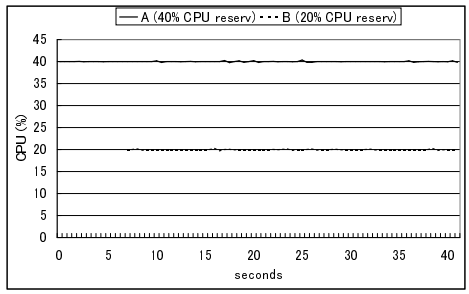

Figure 10. Separate Server, Different Rate (Resource principal A=low request rate, Resource Principal B=high request rate), *No* Network Bandwidth Access Control

with server A is assigned 40% CPU reservation and resource principal associated with server B is assigned 20%.

First we measured the CPU usage without RCLs when server A had low request rate (approximately 12 requests/sec) for a dynamic document and server B had a much high request rate (approximately 80 requests/sec) to the same document. Figure 10 shows the CPU usage of the two principals when RCL was inserted. In both cases, requests to server B start a few seconds after the start of the experiment.

When we have RCLs, we see that the services from server A are not disrupted after requests to server B start. A will always get 40% even if server B is pounded on, and activities associated with server B are contained to 20%. Thus, resource protection with regards to each separate web server is provided with the use of RCLs and reservations. In the next two models, we look at how we can provide the same resource protection capability using one instantiation of the web server.

5.4.1 Single Web Server on Same Machine with Multiple Ports

Here we examine a case where a single server waits on multiple ports, (eg. port 80 and 8080) as shown in Figure 11. We treat all requests coming to port 80 as one resource principal and requests to 8080 as another. To make the implementation of the communication monitor module simple, we identified resource principals based on their IP addresses. We sent requests for dynamic documents to port 80 and 8080 from a group of different IP addresses. In a full implementation, the communication monitor module would need to monitor the destination ports as well as the IP address. In this case, the specification of a resource principal defined by the administrator will contain the port number and group of source IP addresses which can be treated as a single entity. In this experiment, the resource principal associated with port 80 is granted 40% of the CPU resource and port 8080 with 20%.

As Figure 12 shows, when the request rate going to one port is high (80 requests/sec for resource principal B), the resource principal serving the other port with requests at a lower rate (12 requests/sec for resource principal A), can

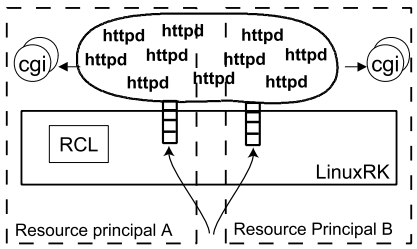

Figure 11. Single web server on same machine with multiple ports

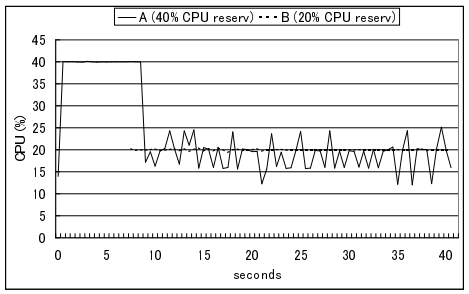

Figure 12. Single web server, Different Port, Different Rate (resource principal A=low request rate, resource principal B=high request rate), *No* Network Bandwidth Access Control

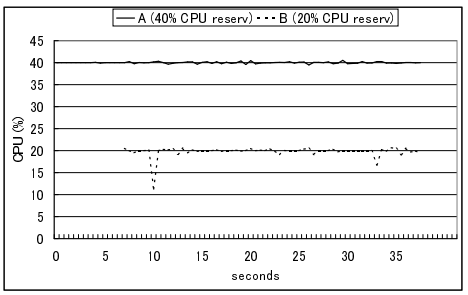

Figure 13. Single web server, Different Port, Different Rate (resource principal A=low request rate, resource principal B=high request rate), *With* Network Bandwidth Access Control

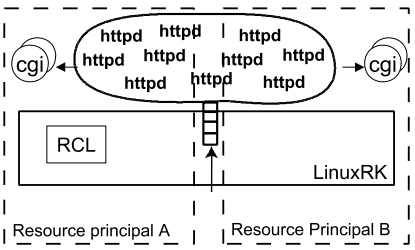

Figure 14. Single web server on same machine with single port

be disrupted even with protection using CPU reservation. We believe this is mainly due to daemons serving the high-rate requests more frequently and then running out of reservations.

One solution is to impose access control on network bandwidth as well. The interesting problem with network bandwidth access control is that we do not necessarily know the absolute value of the bandwidth to grant beforehand. If we wanted to honor the CPU usage, network bandwidth usage depends on the type of requests coming in. If the requests are for a very CPU-intensive dynamic document, then a very small amount of bandwidth should be granted for containment. If we allow larger amounts of network bandwidth reservation for very CPU intensive requests, we see the same disruption similar to the one shown in Figure 12. In our prototype, we inserted an event generator module which monitors the status of each CPU reservation. Based on the events generated when resource reservation is depleted, we adaptively modify the network bandwidth reservation entry in RCL for resource principal B. The result with network bandwidth access control is depicted in Figure 13. We see that the resource usage of resource principal A is not disrupted. There are spikes in the CPU usage for resource principal B. This is due to our adaptation scheme that over-constrains the network bandwidth sometimes. We plan to improve this in the future.

So far, we have shown that we can provide resource protection and the ability to isolate the effect of activities when server ports are not shared using RCLs. By not sharing ports, we implicitly assumed that there is a mechanism to enforce requests from hostile clients to go to different ports, or the clients are cooperative enough to respect this rule. When any requests can be potentially malicious, we must deal with the case where every request goes to the same port.

5.4.2 Single Web Server on Same Machine with Single Port

Here, we look at a case where all requests to the server go through a single port as shown in Figure 14. Resource protection in this model is very important, for instance, at an e-commerce site where requests from browsing customer and shopping customer are treated by the same server and may go through the same port. Or worse, a hostile client may pound on the port of the server providing services to important shopping customers. We want to protect the services provided to the shopping customers and contain the resource used by the hostile client requests.

In this experiment, we classify requests coming from certain IP addresses defined by the administrator as one resource principal A, and other requests from another group of IP addresses as resource principal B. Resource principal A is assigned 40% CPU reservation and principal B is granted 20%. Without network access control, the resource usage of one resource principal can easily be disrupted by traffic to another resource principal. Figure 15 shows the result. We see that as soon as the requests to B starts, A's service is disrupted. The degree of disruption is much greater than the case where resource principals do not share ports. This is because the queue holding the network packets connected to the port is shared. The

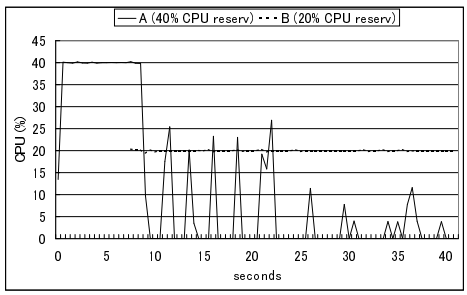

Figure 15. Single web server, Same Port, Different Rate (server A=low request rate, server B=high request rate), *No* Network Bandwidth Access Control

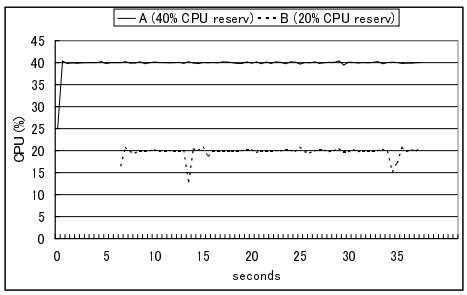

Figure 16. Single web server, Same Port, Different Rate (server A=low request rate, server B=high request rate), *With* Network Bandwidth Access Control

queue is filled mostly with B's requests. To serve A's request that just enters the queue, the server must serve all the requests already in the queue. Figure 16 shows the result with both CPU access control and network bandwidth access control which solves this problem.

Thus, with both CPU and network bandwidth access control, we can contain the effect of unwelcome requests in all three of the models. We have also shown through a series of experiments, unlike receiver live-lock problems reported in [5], relatively small amounts of traffic (that does not come close to saturating the network) can disrupt services if adequate resource protection mechanism is not placed. We have shown that with the use of RCLs, such disruption to services can be controlled.

5.4.3 Discussion

The fundamental difference between QoS provisioning and protection is that QoS provisioning schemes often assume cooperation from applications. In resource protection schemes, not only is it necessary to *guarantee* resources but may also need to *contain* the resource usage of others which may not be cooperative. As we saw in the web server experiments, it is not enough to just make resource guarantees to protect resources. One must make sure that there are mechanisms that guarantees the resources are *actually delivered*. Even if one resource principal was guaranteed 40% CPU and 10kb/sec network bandwidth, if another requested to the server at a much higher rate than 10kb/sec, then that resource principal may not be able to get 40% of the CPU. This implies that CPU reservation and network bandwidth reservation must be applied not only to meet the needs of the principals, but also applied with consideration to the other principals. Thus, implementation of a system-wide resource usage policy is essential.

6 Related Work

Several systems have addressed the protection aspect of resource management. One approach is to restructure the operating system, so resource accounting and enforcement can be efficiently implemented. Nemesis[9] provides operating system functionality as a library. This lets the application and most of the kernel function reside in the same address space. This architecture minimizes cross-talk by isolating data streams into a single address space, and isolates the effect of other applications by making scheduling decisions at each resource multiplexing point implementing an equivalent service to resource reservation in our model. Similarly, the Scout operating system[12, 15] introduces a *path* abstraction representing an I/O channel. The paths are then schedulable and enforceable entities in Scout. This abstraction is similar to the notion of resource principal. Both Nemesis and Scout are operating systems built from scratch to efficiently account and enforce resource usage. Our work focuses on providing fine-grained, and flexible protection policies while preserving the usefulness of general-purpose operating systems.

Numerous systems provide a variant of resource reservation capabilities[19, 16]. Among such systems, several are based on general purpose operating systems[8, 18]. The notion of "resource containers" [1] has been implemented on Digital UNIX. It is pointed out that accountable entities can span multiple protection domains by recognizing them as resource principals independent from protection domains. In their system, it was mostly the responsibility of the application to join or associate itself with the resource principal. (In their scheme, the child processes spawned off by the parent would be associated with the same resource principal. In our scheme, the child process can be automatically associated with another resource principal if so specified in RCL.) The identity of a resource principal can span multiple protection domains in both our system and resource containers, but in resource containers, it is the responsibility of the application to explicitly establish this identity, which assume all the applications are cooperative. We take the approach that the primary identity should be constructed by the system. Thus, we provide a way to dynamically define resource principals and actively enforce an access control policy transparently to the application as opposed to relying on passive privilege checks.

Eclipse [2] provides resource reservation through hierarchical proportional resource-sharing based on FreeBSD. Eclipse represents a hierarchy of resource reservations using the /reserv file system. Eclipse also provides support for legacy applications [3] by putting support for Eclipse-

specific API in their modified version of `libc` library. Files in Eclipse has a new attribute that specifies the resource requirements that will be assigned to the program if the file is read.

More recently, commercial operating systems such as Solaris [17] and AIX [7] have attempted to provide access control facility to resources. Both propose to use crons to statically program the system behavior. To our knowledge, Eclipse and the commercial operating systems mentioned do not support the dynamic definition of resource principals transparently and also address the protection issues. Other commercial products such as ServerXchange[6] support the facilities to provide multiple virtual execution environments on top of a single physical machine. (This is similar to our first Web server model in Section 5.3, Figure 8 where each resource principal has its own instantiation of the web server.) However, our method allows the same application (e.g. web server) to serve multiple resource principals/virtual execution environment.

7 Conclusions and Future Work

In modern operating systems, various system resources, notably files, are managed through some form of access control. In this paper, we 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 proposed and evaluated the use of a new facility called Resource Control Lists (RCLs). There are multiple RCLs in the system to enforce policies at the point where resources are demultiplexed. RCLs enable general-purpose systems with resource reservation capability to control and limit resource usage in a dynamic manner following a specified system-wide resource protection policy.

The definition of resource principals changes over time dynamically and span multiple protection domains. We capture this by using events and resource propagation. One of our future work is to design a more flexible subsystem to securely identify resource principals that originate from the network. We also demonstrated that our facilities can enforce a policy actively and forcefully without explicit cooperation from the applications, even if the applications are hostile. Our empirical results on Linux/RK showed that the overhead imposed by our abstractions is acceptable. We also illustrated how these abstractions can be used by web servers to protect important activities from other potentially malicious activities. We are currently performing extensive evaluation of our facilities, especially focusing on the effectiveness towards defending against full-fledged DoS attacks and intrusion detection and the scalability of RCLs in those environments. We believe RCLs can play a part in a defense system for DoS attacks, cooperating with other components such as routers and firewalls.

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. 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.

[3] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. Quality of Service Support for Legacy Applications. In *Proceedings of The 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, June 1999.

[4] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource Sharing in Reservation-Based Systems. In *The 7th IEEE Real-Time Technology and Applications Symposium (WiP)*, May 2001.

[5] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[6] Ensim ServerXchange",, 2000. http://www.ensim.com/.

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

[8] 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.

[9] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, 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.

[10] 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.

[11] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1998.

[12] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[13] 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.

[14] 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.

[15] O. Spatscheck and L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Feb. 1999.

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

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

[18] 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.

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