# Resource Management for QoS in Eclipse/BSD

J. Blanquer, J. Bruno, E. Gabber, M. McShea, B. Özden, A. Silberschatz, and A. Singh

{blanquer, jbruno, mcshea, eran, ozden, avi, amitsingh}@research.bell-labs.com

http://www.bell-labs.com/what/eclipse

## Abstract

Eclipse/BSD is an operating system derived from FreeBSD. Eclipse/BSD provides flexible and fine-grained Quality of Service (QoS) support for (server) applications. We have implemented hierarchical proportional-share cpu, disk and link schedulers, the /reserv file system providing an API to manipulate "reservations" and a tagging mechanism for the association of reservations with schedulable operations. Currently, Eclipse/BSD is being used to guarantee QoS to server applications, and in particular to differentiate the performance of different web sites hosted on the same platform.

## 1 Introduction

The demand to provide Quality of Service (QoS) guarantees is increasing with the need to run multiple server applications, such as audio and video media servers and web servers, and host services for multiple entities (e.g., companies, individuals) on the same platform. QoS requirements may be client-based, service-based, content-based, and so on. For example, it may be important to differentiate among incoming client requests and to allocate resources based on the client's identity. Systems supporting multiple services, including multiple instances of the same service, may that require system resources be allocated based on server specifications and credentials, leaving it to the servers to differentiate among clients. Many variations are possible and all require the existence of flexible schemes for the management and allocation of system resources such as the cpu, disk and network bandwidth.

In this paper we present, our design and rationale for a flexible, fine-grained resource management API; an implementation of our approach in the Eclipse/BSD operating system; and a set of experiments to demonstrate the soundness of our approach. Our work is based on four key elements: (*i*) the use of hierarchical, proportional-share resource schedulers, (*ii*) the notion of a *reservation* and its implementation in the `/reserv` file system, (*iii*) a tagging mechanism for the association of reservations with schedulable operations, and (*iv*) an access and admission control scheme which leads to a new and useful notion of *reservation domain.*

One of our design goals is to provide QoS support for a large set of (server) applications without imposing significant changes to their design. For example, if an application hosts services for multiple companies, we do not want to require the existence of separate instances of the application, one for each company, in order for them to benefit from QoS support. Another of our design goals is to provide a flexible resource management framework capable of expressing and implementing a large set of provisioning needs.

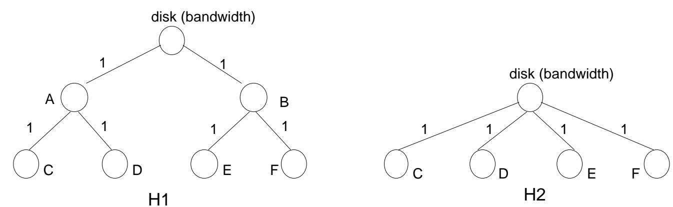


Figure 1: Scheduler hierarchies

### 1.1 Schedulers

Our system makes use of hierarchical, proportional-share resource schedulers [14, 13, 3, 17] at the device driver level for the management of disk and network bandwidth and for cpu scheduling. The resource schedulers we have implemented are dynamically reconfigurable in the sense that the scheduler hierarchy and/or the weights can be altered without stopping the scheduling itself. These reconfigurations are subject to access and admission control, but are otherwise unrestricted.

Hierarchical, proportional-share resource schedulers support a general notion of structured resource *reservations*. Each node in the hierarchy represents a *reservation* with its *share* equal to its weight divided by the sum of the weights of its parent's children (including its own weight). For example, Figure 1 illustrates two scheduler hierarchies H1 and H2 for disk bandwidth. Nodes A and B, in the H1 scheduler hierarchy, each represent a reservation of 50% of the disk bandwidth. At the next level in the H1 hierarchy, each of the nodes, C, D, E, and F, represent a 25% reservation of the overall disk bandwidth, or, in the case of C and D, each has a 50% reservation of A's reservation, etc. It is also interesting to note that in the single-level H2 hierarchy, nodes C, D, E, and F each have a 25% reservation of the overall disk bandwidth. The difference between the H1 and H2 hierarchies is how they dynamically apportion excess bandwidth. For example, if C were not using any disk bandwidth, then under H1, D, E, and F would obtain a 50%, 25%, and 25% share of the disk bandwidth, respectively, while under H2 they would each get a 1/3 share of the disk bandwidth.

We distinguish two kinds of reservation nodes in a scheduler hierarchy: schedulers and (request) queues. The scheduler nodes implement a scheduling algorithm for selecting resource requests from immediate descendant nodes; queue nodes are points where actual resource requests are initially enqueued. Queues are always leaves in the resource scheduler hierarchy. For example, the leaf node C in H1 is a queue at which disk I/O requests may be enqueued. Disk I/O requests and network output requests are represented by `buf{}` and `mbuf{}` headers, respectively.

## 1.2 Reservations

We have designed and implemented the `/reserv` (reservation) file system which provides an API and namespace through which we can access, use, and reconfigure the system resource schedulers. The directory nodes in the `/reserv` file system correspond to the nodes in the scheduler hierarchy and thus represent reservations. We refer to the directories in the `/reserv` file system as *reservation directories* or simply *reservations*. The `/reserv` file system API (described more fully in the next section) provides the means to add and delete reservations and to alter the scheduling weights throughout the hierarchy. Each resource is represented by a reservation directory under `/reserv`: `/reserv/cpu`, `/reserv/wd0` (disk 0), `/reserv/fxp0` (network interface 0), etc. The directory hierarchy under each distinct resource represents all the currently existing reservations for that resource. For example, `/reserv/wd0/r1` and `/reserv/wd0/r2` represent bandwidth reservations `r1` and `r2` on disk 0. Reservations in `/reserv` corresponding to scheduler nodes are called *scheduler directories* and those corresponding to queues are called *queue directories*.

## 1.3 Tagging

An important aspect of our design is the association of a reservation (queue directory) with an operation on an object. The operations on objects include: reading/writing a file, sending a message on a socket, and executing a thread (a process in FreeBSD). The corresponding reservations are disk bandwidth, network interface transmission bandwidth, and cpu cycles, respectively. In our design, we associate reservations with references to objects rather than to the object itself, thereby permitting the object to be shared by different principals without having to share their reservations.

In the case of a file, `f`, we *tag* a file descriptor corresponding to `f` with a reservation. The reservation must be a queue directory for the device on which the file `f` resides. For a connected socket, `s`, we *tag* a file descriptor for `s` with a reservation. The reservation must be a queue directory for the network interface used by packets from `s`. For connected sockets, it is easy to determine the appropriate network interface, but for unconnected sockets, the network interface depends on the destination address which is not known a priori. In this case we provide a mechanism for *late tagging*. We also provide a mechanism for dynamically changing the tag of a file descriptor. In the case of process scheduling in Eclipse/BSD, we *tag* each process with a cpu reservation and provide a means for dynamically changing the tag.

The tags are used to determine the appropriate queue for I/O requests based on tagged file descriptors. For example, if `fd` is a file descriptor tagged with queue directory `q`, then all I/O based on `fd` will be queued at the queue node corresponding to the queue directory `q`. If this were disk I/O, then the `buf{}` structs representing I/O resulting from read/write operations based on `fd` will include a reference to the appropriate queue. Similarly, a tagged process will be be scheduled from the corresponding cpu queue.

### 1.4 Reservation Domains

Access and admission control applied to the `/reserv` file system offer an opportunity to define a useful notion of *reservation domain.* Abstractly, access and admission control are used to restrict (or grant) the right to access, use, and/or reconfigure the `/reserv` file system. Since the `/reserv` file system is the center of all resource management in Eclipse/BSD, the instantaneous set of all rights possessed by a process to use, access, and/or reconfigure the `/reserv` file system is called the *reservation domain* of the process. This notion of reservation domain is independent of the usual notion of a process' protection domain (the importance of this has been argued elsewhere [2]). In our scheme, process credentials include the PID as well as the traditional UID and GID. Access and admission control offer restrictions and rights on the use of queue directories for tagging, the creation of new (sub) reservations, the changing of scheduling weights and/or queue capacities, the setting of garbage collection flags, the passing and/or revocation of rights to other processes (not only related processes), etc. Our notion of a reservation domain is particularly useful in the client-server model where fine-grain control over the use of resources is necessary.

The reminder of the paper is organized as follows. In the following section, we describe the design and implementation of Eclipse/BSD in more detail including descriptions of the resource scheduler interface, the `/reserv` file system, garbage collection, access and admission control, and reservation domains. In Section 3, we present out experimental results. We conclude with a summary of related and future work in Section 4.

## 2 Design and Implementation

In this section, we describe the design an implementation of the resource management system used in Eclipse/BSD. Our current system makes use of hierarchical, proportional-share, dynamically reconfigurable resource schedulers. Reconfiguration includes the ability to alter the resource scheduler hierarchies and read and write scheduler weights and other parameters. A new file system, `/reserv`, has been designed and implemented which provides an API through which applications can configure and use these dynamically configurable resource schedulers. Resources are reserved for OS objects (files, sockets, processes) by associating a reservation (queue directory), not with the object, but rather with a reference to the object. In the case of files and sockets, the reservations are associated with the file descriptors referring to the objects and all I/O based on a particular file descriptor is enqueued at the corresponding scheduler queue.

In order to ease the incorporation of new resource schedulers into Eclipse/BSD, an interface is provided through which resource schedulers can "register" themselves for use under the `/reserv` file system.

### 2.1 Schedulers

Our work is based on the use of flexible resource schedulers. The resources in question include cpu, disks, and network interfaces and each has a resource scheduler. The scheduler for a resource orders the *requests* for access to the resource. Structurally, a *scheduler* is a tree of *scheduler nodes* and *queue nodes*. Queues have no descendants and implement FIFO request queueing (although we do not rule out the possibility of reordering requests within a request queue). Scheduler nodes can have queues and scheduler nodes as immediate descendants. A scheduler node orders the requests emanating from its immediate descendants. The ordering is based on its scheduling algorithm and the (visible) states of its immediate descendants. The *root* of a resource scheduler must be a scheduler node.

All requests for service arriving at a resource scheduler (such as an `mbuf{}` arriving at a network interface) are "tagged" with a reference to a queue. Requests are inserted into the queues corresponding to their tags. The mechanism for assigning tags to requests is an important part or the `/reserv` file system and is described in Section 2.5.

The framework consisting of scheduler and queue nodes supports a wide variety of resource schedulers. For example, a scheduler node with a single descendant queue represents a FIFO scheduler. A scheduler node whose immediate descendants are all queues in which the queues are assigned dynamic "priorities" could implement a priority resource scheduler. Similarly, a scheduler node whose descendant queues are assigned weights could implement a proportional-share scheduler based on weighted fair queueing. Hierarchical resource schedulers are supported by inserting additional scheduler nodes within the hierarchy.

Our current focus has been on hierarchical, proportional share resource schedulers. We have implemented a number of different proportional-share schedulers within

the Eclipse/BSD framework. For the cpu scheduler we use a single-level MTR-LS scheduler [6]; we have implemented hierarchical YFQ for the disk scheduler [4]; and for the network we have implemented the hierarchical $WF^2Q$ algorithm [5].

Our scheduler implementations are novel in the sense that we can add/delete queues and scheduler nodes and change scheduler weights without stopping the resource scheduler. For example, the implementation of $WF^2Q$ maintains an exact GPS simulation while permitting (*i*) the addition and/or deletion of nodes to the scheduler hierarchy, (*ii*) the changing of scheduler weights, and (*iii*) the automatic adjustment of the virtual work and time scales to avoid arithmetic overflow during long-duration busy periods. A detailed description of our $WF^2Q$ implementation is beyond the scope of this paper.

## 2.2 The Resource Scheduler Interface

In this section we describe the interface that a resource scheduler must provide to the kernel in order to "participate" in the `/reserv` file system. The `/reserv` file system interacts with a resource scheduler exclusively through this interface thereby simplifying the task of encorporating new resource schedulers into the `/reserv` file system. Resource schedulers must "register" at boot time by presenting a set of interface routines to the kernel. Registration consists of using `reservfs_register()` to pass pointers to interface routines and an opaque pointer value (void *) to the kernel. The system maintains a Device Translation Table (DTT) which contains registration information for each resource scheduler. The opaque value (denoted by `priv`(ate)) is passed as an argument whenever the kernel uses an interface function. This value can be used by schedulers to identify underlying hardware, such as network interfaces, while using shared code.

Interface functions provided by the resource schedulers to the kernel, using `reservfs_register()`, include:

`init(priv)`: The kernel calls this function very early in the boot process and before any scheduler or queue nodes have been created. The resource scheduler has the chance to allocate and initialize any data structures it needs prior to any calls to `create()`. `priv` is obtained by the kernel during registration.

`create(priv, parent, type)`: A call to this function returns an opaque pointer to a new scheduler node or queue node (determined by `type`, viz., `SCHED` or `QUEUE`, respectively). The `parent` parameter is an opaque pointer to a previously created scheduler node which will be the immediate ancestor of the newly created scheduler node or queue node. If `parent` is null, then the newly created (scheduler) node is considered the *root* scheduler node of the resource scheduler.

`start(priv, root)`: This function is called by the kernel after all initial nodes have been created (at least the `root` scheduler node and a queue) and is used to start the scheduling of tagged requests for this resource. The resource scheduler responds to this function by enabling handlers or enabling the underlying controller to respond to events and by commencing to schedule tagged requests for the resource.

`delete(priv, node)`: This function is called to delete `node` where `node` is an opaque pointer to either a scheduler or queue node. Immediate deletion may require flushing outstanding requests. If the resource scheduler does not support immediate deletion (for example the network scheduler may continue to send previously enqueued packets), the resource scheduler could indicate successful deletion even though the actual deletion takes place after all pending requests have been serviced. Otherwise, the function could return failure and, in this case, the kernel will call `delete` again at a later time.

`get/set(priv, node, values, type)`: These functions allow the kernel to get/set scheduler parameters. The API interprets `values` differently depending on the resource scheduler, the `type` argument, and the `node` reference (queue or scheduler node). The result of a set operation may fail due to admission control checks.

The `create`, `delete`, and `set()` interface functions must not interfere with the resource scheduler's operation. It is assumed that these calls can be made while the resource scheduler is running.

## 2.3 The /reserv API

In this section we describe `/reserv` API and its use of the interface to the resource schedulers. Due to the hierarchical nature of the resource schedulers, it is natural and convenient to associate a directory hierarchy with the scheduler and queue nodes of the resource schedulers.

The directory hierarchy of the `/reserv` file system consists of two types of directories, viz., *scheduler directories* and *queue directories*. Since scheduler and queue directories correspond to resource reservations, we will sometimes refer to them as *reservation directories*

or simply as *reservations*. Scheduler directories correspond to scheduler nodes and queue directories correspond to queue nodes. Each independently scheduled resource is represented by a scheduler directory under /reserv which corresponds to the root scheduler node for that resource. For example, we have /reserv/cpu, /reserv/wd0, and /reserv/fxp0 corresponding to the cpu, disk, and network resources, respectively.

Every scheduler directory contains (among other scheduler and queue directories) files named share, newsched, and newqueue. Each queue directory contains two files named share and backlog. Files (scheduler directories, queue directories, share, backlog, newsched, and newqueue files) in /reserv are represented by reserv{} nodes. The reserv{} structure has a type field and other scheduler-independent information regarding the file, much like the inode{} of a traditional Unix file system. The reserv{} structures representing scheduler directories and queue directories contain opaque pointers to scheduler and queue nodes, respectively. These fields hold the values returned by calls to create().

In the following paragraphs, we explain the use of the various /reserv files. Let dir be a reservation, either scheduler or queue, and pdir be its parent reservation. The file dir/share (in the case of a proportional-share scheduler) contains two values: the weight with which dir shares the resource, available from pdir, among its siblings and a minimum guaranteed absolute value. Writes to dir/share attempt to change these values and are subject to admission control. If pdir is /reserv, then the dir/share file is read-only and represents the resource in its entirety.

If dir is a queue directory then reading from dir/backlog returns the count of requests served and the aggregate quantity of service provided (in cpu time or I/O bytes). Writing to dir/backlog sets the maximum number of requests and aggregate quantity of service that may be enqueued, and resets the corresponding counters.

The newsched and newqueue files are used to create new scheduler and queue directories. Opening dir/newsched creates a new scheduler directory in dir along with its share, newsched, and newqueue files. During the open, a call is made to the create(priv, parent, SCHED) function which returns an opaque pointer to a new scheduler node. The value of parent is equal to the value of the opaque pointer stored in the reserv{} node representing dir. The names of scheduler directories begin with r (for reservation) and within dir are numbered uniquely. For example, dir/r0, dir/r1, etc. The file descriptor of the share file in the newly created scheduler directory is returned by the open call.

Similarly, by opening dir/newqueue, a new queue directory is created in dir along with its corresponding share and backlog files. In this case a call is made to create(priv, parent, QUEUE) which returns an opaque pointer to a new queue node. The names of queue directories begin with q and, within dir, are uniquely numbered. The file descriptor of the new share file is returned.

In the Section 2.5 we describe the connection between queue directories and schedulable requests on OS objects such as files, sockets, and processes.

## 2.4 /reserv Initialization

Initialization of /reserv occurs after the resource schedulers have registered and before any file system is mounted. The kernel calls init(priv) for every resource scheduler registered in the DTT—this gives the schedulers a chance to allocate and initialize any needed internal data structures. Next the kernel creates a reserv{} node for the root directory of the /reserv file system. Then it creates two reserv{} nodes for each resource scheduler, one for the *root* scheduler directory and one for a queue directory in the root scheduler directory. In addition to these, the kernel creates reserv{} nodes for the share files in each directory, the backlog file in the queue directory, and the newqueue and newsched files in the scheduler directory. For each resource scheduler, the kernel calls create(priv, NULL, SCHED) to create the root scheduler node, root, and calls create(priv, root, QUEUE) for a second time to create a queue node. Figure 2 shows the data structures for the root of the /reserv file system, a scheduler directory, a queue directory, and the rest of the files that are created during initialization. Each of the queue directories (once mounted under /reserv) is named q0, that is, /reserv/cpu/q0, /reserv/wd0/q0, etc.

Once the scheduler and queue directories have been created for each resource scheduler, start(priv, root) will be invoked to commence scheduling. This could involve activating the hardware or switching over from a standard FreeBSD scheduler to the registered resource scheduler. At this point the kernel is able to use the queue nodes of all the schedulers even before

the `/reserv` file system is mounted. Initial daemons launched from "rc" scripts will use these request queues.

We will come back to how this is accomplished after we have discussed *default lists.*

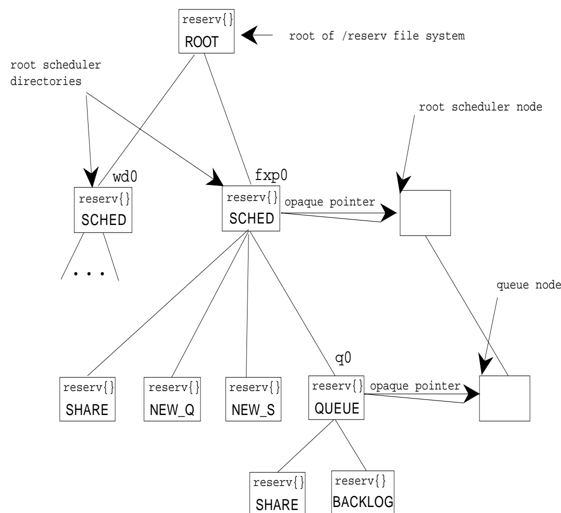


Figure 2: /reserv file system

Access to the `/reserv` file system API is through the FreeBSD `vnode/vfs` file system interface. The `/reserv` file-system dependent data corresponds to the `reserv{}` nodes and the the file-system dependent `vnodeops` and `vfsops` have been implemented to support the `/reserv` API. File system operations cannot take place until the `/reserv` file system is mounted. After system initialization, a `vnode{}` is created for the root of the `/reserv` file system and it is mounted on `/reserv`. From this point on, the `/reserv` API is available.

## 2.5 Tagging and the Default List

Every request arriving at a resource scheduler must be *tagged* with a pointer to a queue node. These are the opaque pointers returned by calls to `create(priv, parent, QUEUE)`. Thus I/O data structures, such as `buf{}` for disks and `mbuf{}` for network output, gain a tag field to hold an opaque pointer. We have also augmented the `uio{}` structure with a new tag field which is used to propagate opaque pointers to lower levels for the ultimate tagging of I/O requests. At the resource scheduler, the tag (opaque pointer) is used to quickly locate the queue to which the request belongs.

Earlier we have discussed the use of tags when tagging file descriptors and processes. These tags were references to queue directories. In this discussion, a *tag* can be either a pointer to a `vnode{}` corresponding to a queue directory or a (opaque) pointer to a queue node. If the tag is a pointer to a `vnode{}`, then the `vnode{}` holds the pointer for the corresponding `reserv{}` node and the `reserv{}` node holds the corresponding tag, the opaque pointer to queue node.

Since reservations (scheduler or queue directories) are garbage collected when they are no longer being used, we need a way to keep track of references to reservations. Garbage collection, under Eclipse/BSD, is triggered by detecting when the number of references to a reservation goes to zero. We have chosen to use the `vnode{}` reference count to hold the count of the total number of references to the corresponding reservation. In what follows we point out the new locations for `vnode{}` tags that have been introduced in Eclipse/BSD. These additional references to `vnode{}`s are accounted for in the `vnode{}` reference count field.

A new tag field is associated with each file descriptor. The `proc{}` structure has been expanded with a tag field for its cpu reservation. These tags are pointers to vnodes.

The *default list* of a process is a list of `vnode{}` tags, one for each resource. The default list of a process is used to provide tags when a tag is not otherwise specified. The default list of a process `PID` is represented as a new file `/proc/PID/default` which contains the tags. Reading the `default` file returns the names of the of queue directories corresponding to the tags, e.g., `/reserv/cpu/q2` `/reserv/wd0/r1/q0` `/reserv/fxp0/r5/q3`. A process, with the appropriate access rights, can write the `default` file to change entries. There is also a second list of tags, called the *child default-list*. The child default-list is also represented as a file, i.e, `/proc/PID/cdefault`. When a process forks, the child's default list and child default-list are set to the child default-list of its parent.

When a process uses any of the `open()`, `accept()` or `connect()` system calls, the kernel will automatically tag the returned file descriptor with the tag for that resource currently listed in the default list of the process. For example, when a process issues a `connect()` call, the kernel determines the network interface, locates the registered resource scheduler for that interface, and tags the returned file descriptor with the tag for that interface currently appearing in the process's default list. In gen-

eral, a file descriptor's tag is initialized for vnodes{} at open time; for connected sockets at connect or accept time; for unconnected sockets at sendto or senddmsg time. This latter case is referred to as *late tagging* since, even if the file descriptor is tagged, the message destination may require a network interface that is different than the one corresponding to the tag. So even if the file descriptor of a unconnected socket is tagged, we must check whether it can be used. If not, the appropriate tag from the default list is used.

The tag of a file descriptor can be explicitly set or read by a new fcntl system call with the new commands: F_SET_QUEUE, for explicitly setting a file descriptor's tag, and F_GET_QUEUE, for obtaining the name of the queue directory corresponding the the tag currently associated with the file descriptor. In the case of F_GET_QUEUE, if the file descriptor corresponds to a file in /reserv, the the pathname corresponding to the file is returned. System call chcpures() has been added to change the tag corresponding to a process's cpu reservation.

The file descriptor tags are used in the read(), write(), sendto() and sendmsg() system calls to determine the queue tag (opaque pointer) for the request entity (buf{}s and mbuf{}s) that arrives at the resource scheduler. Thus we are assured that all requests arriving at a resource scheduler are tagged.

At boot time, after the start(priv, root) function has been called and before the /reserv file system is mounted, the default list of the init process contains tags to each of the initial queue directories created for each resource (init's child default-list is set to the default list). This means that the necessary vnode{}s have been allocated but are not enqueued in the file system vnode queue because the /reserv file system is not mounted. However, even though the /reserv file system is not yet mounted, the tagging mechanism, based on vnode{} tags and the default list, is operational and therefore requests arriving at resource schedulers will be tagged. When /reserv is mounted, the previously created vnodes will be transparently enqueued in the file system vnode queue with reference counts appropriately set.

## 2.6 Garbage Collection

The /reserv file system uses garbage collection to delete reservations (scheduler and queue) that are no longer referenced. In our design, the vnode{} reference count accumulates all references to reservations in the /reserv file system, i.e., tags and other kernel vnode{} references. References to /reserv files occur in file descriptors with tags, in the default list and child default-list of processes, in the file{} nodes corresponding to open files in /reserv, and current directory references for processes in the /reserv file system. The vnode{} reference count is used to trigger garbage collection and provides a clean triggering mechanism through the inactive() call of the vnode interface.

A flag, GC, in the reserv node indicates whether the node should be garbage collected when inactive() is called. When a reserv{} node is created, GC is enabled, but the flag can be reset using new commands to the fcntl system call. These commands, F_SET_COLLECT and F_GET_COLLECT, can be used on file descriptors for reservations in the /reserv file system to set or get the GC flag.

Although, in our current implementation, garbage collection is triggered by the vnode{} reference count, we can envision other policies including those based on expiration duration, expiration time, etc.

When the garbage collector is triggered from the vnode's inactive() function, it will try to delete the subtree rooted at that vnode{} and then collapse the tree upstream towards the root. Garbage collection uses the locking semantics implemented in the /reserv file system. Locks on the reserv{} nodes are needed to maintain atomicity on access and also between a parent and its children. For example, the locks are used to serialize two or more processes attempting to allocate a vnode{} for the same reserv{} node, or to avoid two garbage collectors working on the same nodes at the same time (kernel threads). Even though this latter case does not arise in the current implementation, we are anticipating the need in order to ease the ports to newer FreeBSD versions or other Unix systems.

The garbage collector will collect a reserv{} node if its corresponding vnode{} reference count is zero and it is not locked (directly or indirectly by its immediate parent). Prior to deleting a reserv{} node, the resource scheduler's delete() function is called on the corresponding *scheduler* or *queue* node. If this call does not succeed, then the garbage collection at that vnode{} fails and has to be retried later. To provide an asynchronous interface, the scheduler returns success while marking its own node for deletion after having serviced the pending requests. We have implemented both styles

of deletion.

## 2.7 Access Control

Access control constrains the ability of a process to use and/or modify the objects belonging to the `/reserv` file system. Processes are endowed with "credentials" which are used to determine whether certain operations are allowed. The standard Unix credentials of user and group ids are not sufficient to build a useful access control mechanism for the `/reserv` file system. One reason is that many servers create subprocesses, all with the same credentials, to deal with clients that are not all granted identical access to resources. At the very least, a processes PID needs to be part of its credentials. In general, other kinds of unforgeable credentials may prove to be useful.

Below we identify the Read (R) and Write (W) operations on `/reserv` which is subject to access control:

*scheduler directory*: R: reading the directory and getting the `GC` flag; W: writing the directory and setting the `GC` flag.

*queue directory*: R: Reading the directory, using the directory as a tag, and getting the `GC` flag; W: writing the directory and setting the `GC` flag.

*share and backlog files*: R: reading; W: writing subject to admission control.

*newqueue and newsched files*: R: opening; W: opening.

Abstractly, an access control policy is represented by a matrix, ACC, of subjects (credentialed processes) and objects from `/reserv` (e.g., scheduler directory, share file, etc.). The entry denoted ACC[S, O] is the list of operations allowed by subject S on object O. Finally, there is a list of processes associated with each entry of ACC[] designating those processes which are allowed to modify the entry and to what extent.

The access rights (with respect to `/reserv` of a process with credential S are represented by row S of the access matrix ACC. The operations specified in ACC[S, .] are exactly those which are available to S (subject to additional admission control restrictions). Conceptually, we identify the *reservation domain* of a process with its corresponding row in ACC. Our initial implementation is based on the use of capability lists associated with `/reserv` entities.

## 2.8 Admission Control

Admission control deals with requests which are allowed or disallowed based on resource limitations. In our current implementation we make extensive use of proportional-share resource schedulers. Let `dir` denote a scheduler directory and `d1`, ..., `dn` its immediate subdirectories (scheduler or queue directories). Let `sj` denote the `share` file for directory `dj`. `sj` specifies two quantities a weight, $w_j$ and a minimum absolute amount, $m_j$, of the resource.

The weights determine how the resource, reserved for `dir`, is divided up among its immediate descendant directories. Specifically, the fraction of `dir`'s resource reserved for `dj` is equal to $w_j / \sum_{j=1}^{n} w_j$. Suppose $m$ is the minimum amount of resource guaranteed to `dir` (specified in `dir`'s `share` file). Then the condition $m_j \leq w_j m / \sum_{j=1}^{n} w_j$ is checked by admission control for all directories which may be affected when writing to a share file. Such writes are admissible if and only if this condition will not be violated at any reservation in `/reserv`.

Writing to a `backlog` file is also subject to admission control since memory resources are involved.

## 2.9 Reservation and Reservation Domains

A reservation corresponds to a node in a resource scheduler and thus represents a "share" of the corresponding resource as provided by the scheduling algorithm of the resource scheduler. The `/reserv` API enables us to organize and structure reservations (resource schedulers): we can create new scheduler and queue directories (using the `newsched` and `newqueue` files), change resource scheduler parameters (writing to `share` files), and alter buffer capacities (writing to `backlog` files).

A queue directory represents a queue node in a resource scheduler and resource requests entered into this queue will obtain the QoS which has been provisioned for this queue node. Our tagging mechanisms enable us to associate queue directories with file descriptors and processes and thus provide QoS support for operations on these objects. The assignment of reservations to objects is quite flexible. Reservations can be shared by more than one object (the same tag can be associated with file descriptors representing different files) and a file can be associated with more than one reservation (by tagging file descriptors to the same file with different tags).

Tags can be assigned automatically using the default list or explicitly using `fcntl` and the garbage collector will automatically collect unused reservations. The use

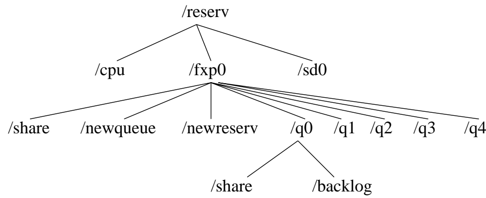


Figure 3: The state of /reserv directory used in the experiments.

of default lists and garbage collection makes it possible to provision resources for legacy applications. Reservations can be assigned to the default list of an application without its knowledge and it will transparently obtain the QoS support provided by these reservations. When the application finishes, these reservations can be transparently garbage collected.

The reservation domain of a process determines its ability to manipulate and use the `/reserv` file system. It is a very general notion and complements the traditional idea of a protection domain.

# 3 Experiments

Eclipse/BSD is derived from FreeBSD [11]. It employs the MTR-LS, YFQ and WF$^2$Q proportional-share schedulers for cpu, disk and output link, respectively, and Signaled Receiver Processing (SRP) for network input processing [6, 4, 5]. SRP demultiplexes incoming packets before network and higher-level protocol processing, and processes input protocols in the context of the respective cpu reservations. The Eclipse/BSD implementation used in these experiments adds 10884 lines of code to FreeBSD version 2.2.8: 3230 lines for the `reserv` file system and modifications to the `proc` file system, and 7654 lines for the new schedulers and their integration into the kernel. The kernel size in the GENERIC configuration is 1601351 bytes for FreeBSD and 1643939 bytes for Eclipse/BSD (an increase of only 43.9 KB). The Eclipse/BSD code will be available at http://www.bell-labs.com/what/eclipse.

In our previous papers [6, 4, 5], we presented experiments verifying that Eclipse/BSD's cpu, disk and network schedulers provide QoS guarantees, including isolation and minimum bandwidth. This section demonstrates experimentally the effectiveness of Eclipse/BSD framework to provision system resources, and in particular, to change reservations on the fly (e.g., the cpu reservation of a process and output link reservation of a socket) and to share reservations among applications.

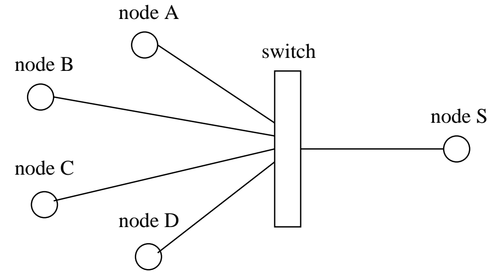


Figure 4: Node S is a Web server that hosts multiple sites on either FreeBSD or Eclipse/BSD.

We consider a server where Web sites of four companies are hosted, and the server's administrator wants to isolate the performance of services provided to these four companies. That is, when a system resource is overloaded, the service for a company should not be affected by the load on other companies. The administrator configures the `/reserv` file system as illustrated in Figure 3. The queue directories `q1`, `q2`, `q3` and `q4` in `/reserv/fxp0` contain the same files as `q0`, and the reservation directories `/reserv/cpu` and `/reserv/sd0` have the same structure as `/reserv/fxp0`. Each queue directory `q1`, `q2`, `q3` and `q4` have a weight of 0.24 and corresponds to a different company, whereas `q0` has a weight of 0.04 and corresponds to the shared activities on the server.

We ran experiments on the configuration shown in Figure 4, where HTTP clients on nodes A to D make requests to the HTTP server on node S. Nodes A to D are Pentium Pro PC's running FreeBSD. The operating system varies only in node S, being either FreeBSD or Eclipse/BSD. Node S is a PC with 266 MHz Pentium Pro CPU, 64 MB RAM, and 9 GB Seagate ST39173W fast wide SCSI disk. All nodes are connected by a Lucent P550 Cajun Ethernet switch at 100 Mbps. Node S runs the Apache 1.3.3 HTTP server and hosts multiple Web sites. Nodes A to D run client applications (some derived from the WebStone benchmark) that make requests to the server. All measurements are the averages of five three minutes runs.

The Apache 1.3.3 HTTP server for FreeBSD is a multi-process server. It (pre)forks upto a predefined number of processes. Each HTTP connection is handled by a process. The number of requests a process services before it terminates can be specified. Idle and/or new processes accept new connections from the listening socket(s).

## 3.1 Isolating Performance of Web Sites

Apache supports a number of ways to host multiple Web sites on a single server, including running a separate Apache server for each Web site, or running one Apache server configured with multiple *virtual hosts* one for each site. For each of these cases, Eclipse/BSD provides an effective framework to isolate and differentiate performance of Apache Web sites and provide them with performance bounds. In our previous papers, we demonstrated the QoS capabilities of Eclipse/BSD when each Web site is hosted with a separate Apache server. Here, we experiment with a single Apache server configured with four virtual hosts each corresponding to a different company's web site. These experiments demonstrate the need to modify the reservations of a process, socket and file descriptor on the fly.

In order to guarantee QoS to each virtual host, when an Apache process accepts a new request from a listening socket, the cpu reservation of the process needs to be set to the one corresponding to the virtual host, at which the request arrived. Similarly, the network reservation of the connected socket needs to be set to the one corresponding to the virtual host (for brevity, we omit reservation and scheduling of other resources, such as disks since experiments in this paper only overload cpu and output link). To do so, we have written an Eclipse/BSD module (`mod_eclipsebsd` in Apache parlance) that enables Apache to utilize the QoS support of Eclipse/BSD. Apache uses the module concept to extend its functionality (without the need to change the Apache source tree). As of Apache 1.3., it is possible to compile the Apache core into a dynamic shared object library, and thereafter compile Apache modules as shared objects that can be dynamically loaded at run-time.

The `mod_eclipsebsd` module adds new configuration directives to specify the resource reservations corresponding to a virtual host. Once an Apache process accepts a request, `mod_eclipsebsd` sets the cpu reservation of the process and the output link and disk reservations of the file descriptors of the process appropriately.

### CPU Intensive Workload

In the first experiment, an increasing number of clients continuously made CGI requests to either of four Web sites hosted at node S. Processing of each of these CGI requests consists of computing half a million random numbers (using `rand()`) and returning a 1 KB reply. Therefore, the bottleneck resource is the cpu. We kept

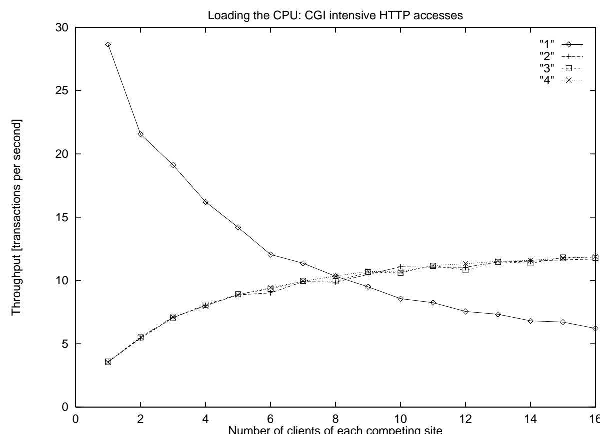


Figure 5: FreeBSD cannot provide isolation to different sites.

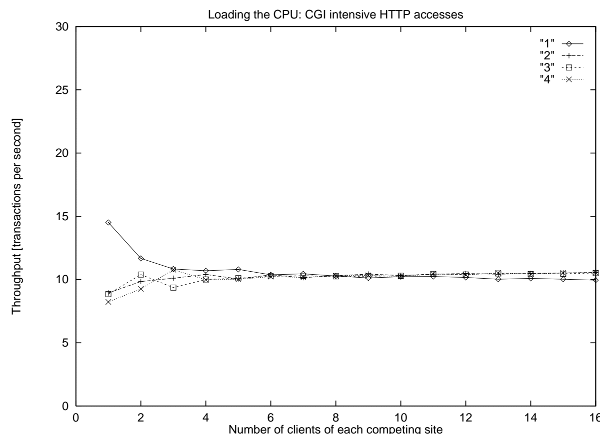


Figure 6: Appropriate CPU reservations in Eclipse/BSD can guarantee isolation and/or a minimum throughput for each site independent of the load at other sites.

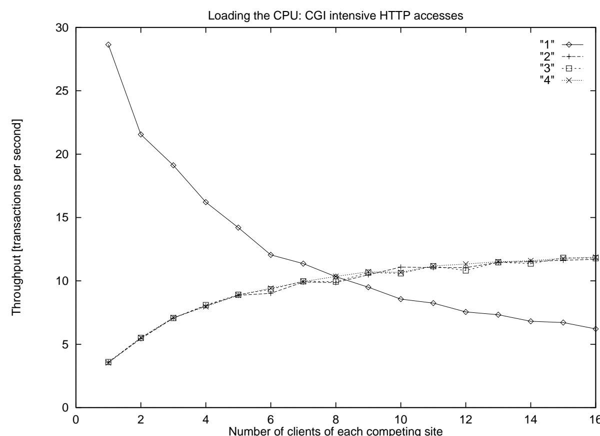


Figure 7: The performance of Eclipse/BSD without reservations is similar to the one of FreeBSD.

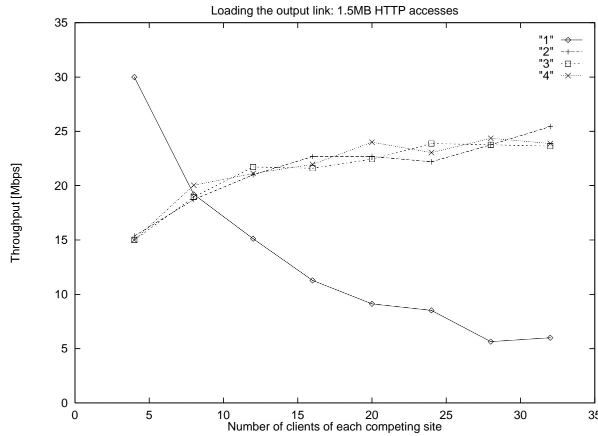


Figure 8: FreeBSD cannot provide isolation to different sites.

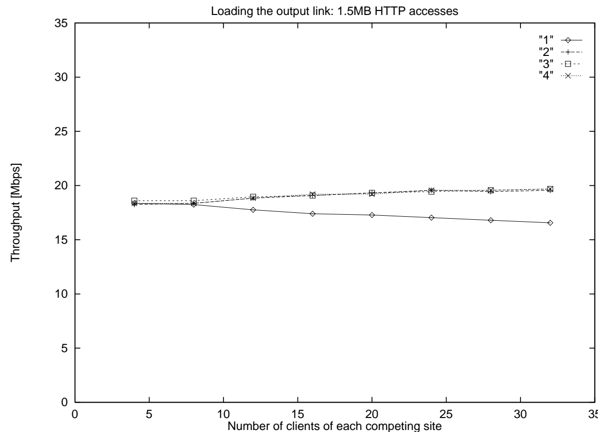


Figure 9: Appropriate reservations in Eclipse/BSD can guarantee isolation and/or a minimum throughput for each site independent of the load at other sites.

the number of concurrent clients to site 1 at eight while we varied the number of concurrent clients to other sites. Figures 5, 6 and 7 show the average throughput (CGI transactions per second) of each site under FreeBSD, under Eclipse/BSD when Apache is configured with mod_eclipsebsd module that sets the reservations, and under Eclipse/BSD when Apache is not configured with mod_eclipsebsd.

In the latter case, each process runs under the default reservation on each resource (i.e., in our setup under q0's). Performance is roughly the same as performance on FreeBSD. In both cases, each process obtains equal treatment. Thus, the throughput of a site depends on the ratio of the number concurrent requests for the site over the total number of concurrent requests at the server. As a result, the sites cannot be isolated. On the other hand, under Eclipse/BSD when Apache server is configured with mod_eclipsebsd, each site gets at least one quarter of the resources, when the number of concurrent requests for a site reaches a few. Thus, it is possible to isolate the performance of sites.

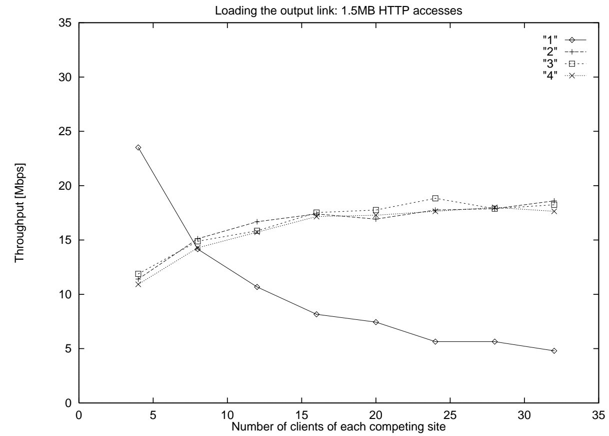


Figure 10: The performance of Eclipse/BSD without reservations is similar to the one of FreeBSD.

### Output Link Intensive Workload

In the next experiment, an increasing number of clients continuously requested the same 1.5 MB document from either of four Web sites hosted at node S. Given that requests are much smaller than replies, little processing is required per request, and the requested document fits easily in the node S's buffer cache, the bottleneck resource is S's network output link.

We kept the number of concurrent clients to site 1 at eight while we varied the number of concurrent clients to other sites. Figures 8, 9 and 10 show the average throughput of each site under FreeBSD, under Eclipse/BSD when Apache is configured with mod_eclipsebsd module that sets the reservations, and under Eclipse/BSD when Apache is not configured with mod_eclipsebsd. Results are very similar to those of CGI workload. We see that even with reservations, the throughput of site 1 is (slightly) impacted by the load at sites 2, 3 and 4. This is because that clients for each site are distributed to nodes A, B, C and D in a round-robin manner, and they are impacting each other's performance (remember that nodes A, B, C and D are running FreeBSD without any QoS support).

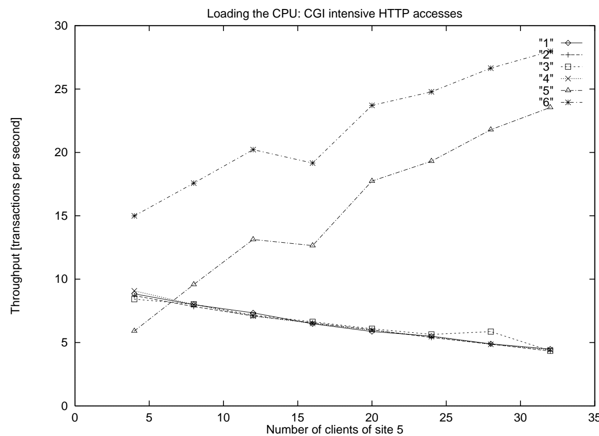


Figure 11: FreeBSD cannot provide isolation to different companies.

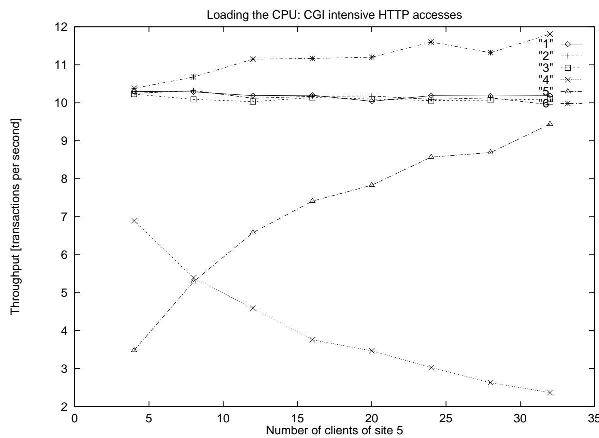


Figure 12: Appropriate reservations in Eclipse/BSD can guarantee isolation and/or a minimum throughput for each company independent of the load at other companies.

## 3.2 Isolating Performance of Reservations

The reservation domain of a process determines its ability to manipulate and use the `/reserv` and `/proc` file systems through which it is able to create and provision reservations, and tag files, sockets, and processes with reservations. Eclipse/BSD enables processes to share reservations without the need to share reservation domains or have a hierarchical relationship among reservation domains. This provides a flexible framework for expressing many resource provisioning needs. The following examples illustrate the effectiveness of our framework. Consider the case where different resources are to be provisioned differently among two companies: The companies need to share a cpu reservation, but have separate output link reservations. Application A is hosting services for company 1 and another application B is hosting services for company 2. Application A and B can share a cpu reservation without the need to share output link reservations. Consider another case where resources are to be provisioned among three companies. Application A is hosting services for companies 1 and 2, and application B is hosting services for companies 2 and 3. Application A and B can share reservations for company 2 without the need to share reservations for other companies. Also, our framework enables us to distinguish between the reservation domain of a process and the traditional protection domain.

For the next experiment, we consider a scenario in which the system administrator wants to provision the system equally among four companies independent of services supported for the companies. Now, assume that the system needs to host the external Web pages of these companies, as well as a separate Web site for company 4 consisting of some "sensitive" data. The administrator wants to run a single Apache server (with a separate virtual host for each company) to service the external web sites of these four companies respectively called site 1, 2, 3 and 4), and a separate Apache server for the second Web site of company 4 (this site is called site 5). The desired provisioning can be done in Eclipse/BSD by using the `/reserv` in Figure 3 and giving permission to Apache server for external pages to queue its request under any of the request queues and to the second Apache server to queue its request only under `q4`.

### CPU Intensive Workload

Again, an increasing number of clients continuously made CGI requests to either of five Web sites hosted at node S. We use the same CGI script as in Section 3.1 to overload

the cpu. We kept the number of concurrent clients to each of the external sites at eight while we varied the number of concurrent clients to the second Apache server (site 5).

Figures 11 and 12 show the average throughput (CGI transactions per second) of each site and average aggregate throughput of sites 4 and 5 (graph 6 corresponding to company 4) under FreeBSD and under Eclipse/BSD when the first Apache server is configured with `mod_eclipsebsd` module.

Eclipse/BSD provisions resources equally among the four companies. The load increase on the fourth company's second Web site (site 5) does not degrade the performance of the other companies' Web sites, only impacts the performance of the fourth company's external site (site 4); but the aggregate throughput provided to company 4 remains at least one fourth of the available throughput. On FreeBSD, the performance of all companies' external Web sites suffer with the load increase at company 4's second Web site (site 5). Thus, the throughput of services provided to companies cannot be isolated.

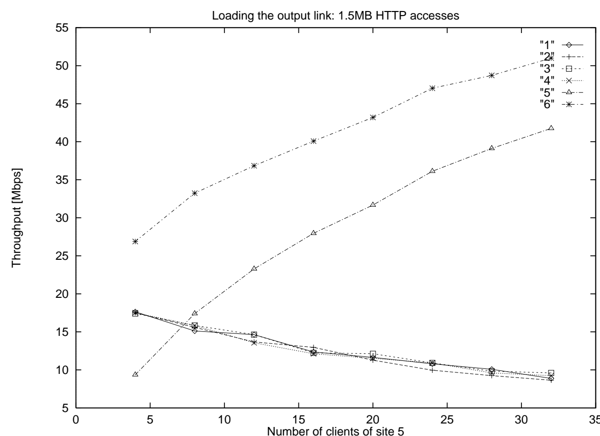


Figure 13: FreeBSD cannot provide isolation to different companies.

**Output Link Intensive Workload**

In the next set of experiments, we use the same output link intensive workload described in Section 3.1. Thus, an increasing number of clients continuously requested the same 1.5 MB document from all the Web sites hosted at node S. We kept the number of concurrent clients at the external Web sites at eight, while the number of concurrent requests at the second Web site of fourth company (site 5) is increased.

Figures 13 and 14 show the average throughput of each site and average aggregate throughput of sites 4 and 5 (graph 6 corresponding to company 4) under FreeBSD and under Eclipse/BSD when the first Apache server is configured with `mod_eclipsebsd`. Results are very similar to those of CGI workload.

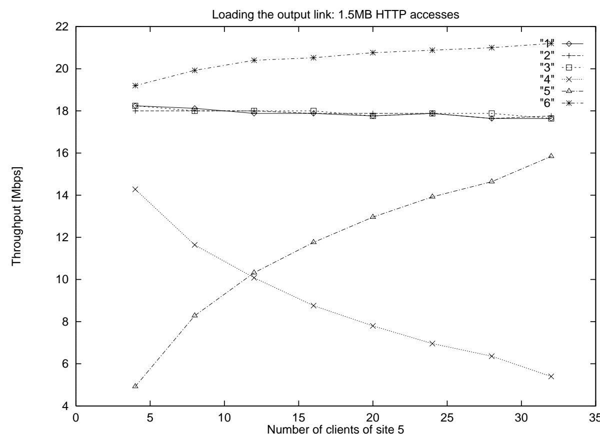


Figure 14: Appropriate reservations in Eclipse/BSD can guarantee isolation and/or a minimum throughput for each company independent of the load at other companies.

## 3.3 Overhead of New Primitives

Two new system calls are extensively used in these experiments: `chcpures()` that changes the cpu reservation of a process and `fcntl()` with F_SET_QUEUE option that changes the reservation of a file descriptor. The overhead of these system calls is presented in Table 1. For each measurement, we wrote a user-level program that invoked the corresponding system call 10,000 times, measured the total elapsed-time, and calculated the average overhead of the system call. The first row in Table 1 is the baseline

| Operation | Overhead [$\mu$s] |
|---|---|
| null syscall | 1.85 |
| chcpures | 5.64 |
| fcntl(F_SET_QUEUE...) | 9.99 |
| creating a queue directory | 10.28 |

Table 1: Overhead of new primitives used in the experiments

overhead of a system call in Eclipse/BSD and FreeBSD running on node S (described for Figure 4).

We also measured the overhead of creating (i.e., opening the corresponding `/newqueue` file) queue directories under `/reserv` file system. The overhead of this operation depends on the size and the shape of the `/reserv` file system and the resource (remember that creating a queue directory also creates a scheduler queue). The number listed in this table is the average of creating 10 queue directories under `/reserv/cpu`.

# 4 Related and Future Work

The literature in scheduling research is quite extensive, especially for packet scheduling [14, 13, 3, 9, 17]. Our use of hierarchical, proportional-share schedulers borrows heavily from this body of research. We have added a dynamic element to these schedulers since we require that they be reconfigurable while running.

The scheduler model we have adopted is well-suited for simple proportional-share schedulers. We are interested in pushing this model to include more real time dependencies. More general scheduler models such as *decoupled generalized processor sharing* [18] and *service-curve based schedulers* [8] are available and we would like to extend Eclipse/BSD in these directions. The Eclipse/BSD code will be available at http://www.bell-labs.com/what/eclipse.

ALTQ [7] provides a network queueing framework (API) for output link scheduling that allows the use of several queuing disciplines. Like Eclipse/BSD, ALTQ is implemented as an extension to FreeBSD. The ALTQ API simplifies the task of including new network schedulers into FreeBSD and encourages researchers to deploy their schedulers under this framework. We were unable to make use or the ALTQ framework, since the Eclipse/BSD schedulers provide a significantly different interface to the kernel, one that supports dynamic reconfiguration within the scheduler.

The API in Eclipse/BSD is based on the Unix file system. It is not unusual in Unix to extend the file system namespace [15]. The `/reserv` file system gives us a convenient and uniform way to deal with reservations. Other systems, especially those that emphasize real-time scheduling, have not extended the file system to cover the management of resources:

Rialto's [10] real-time distributed framework takes a user centric approach for resource management in which applications negotiate the desired QoS parameters with a resource planner object. Resource provider objects roughly correspond to our resource schedulers. The Rialto approach is object oriented and disposed towards real-time.

The work done in on *processor capacity reserves* [12] also approaches the QoS management from a real-time perspective and deals with microkernel-specific issues. Their *processor capacity reservations* may be used by different threads allowing multiple user level servers to participate in one "reservation."

*Resource kernels* from CMU [16] have evolved from the work on Processor Capacity Reserves. This work is also real-time oriented and reservations are expressed in terms of processor requirements per time period. Their work addresses the priority inversion problem between different reservations. They identify the problem of processor *co-dependency* and deal with resource composability.

The work in [1] shows that by adding priorities to service requests on a web server, the server can provide differentiated QoS. Rather that building QoS into the kernel, they map requests to two different numerical priorities delivering them accordingly in order, to web processes.

The research in [2] introduces *resource containers* as the focus of OS resource management. They address the issue of decoupling protection domains from resource domains. Their resource container abstraction combines aspects of our reservation domains and default lists. However, we have found that separating the notion of reservation domains from the reservations provides more flexible fine-grain resource management.

# References


[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. "Providing Differentiated Quality of Service in Web Hosting Services". In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the USENIX 3rd Symposium on Operating System Design and Implementation New Orleans, LA, October 1999*, February 1999.

[3] Jon C. R. Bennett and Hui Zhang. "Hierarchical Packet Fair Queueing Algorithms". In *Proceedings of the ACM SIGCOMM California, August 1996*, August 1996.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with Quality of Service guarantees. In *ICMCS99*, June 1999.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting Quality of Service into a time-sharing operating system. In *Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, California*, June 1999.

[6] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse operating system: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, Louisiana*, pages 235–246, June 1998.

[7] K. Cho. A framework for alternate queueing: Towards traffic management by pc-unix based routers. In *Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, Louisiana*, June 1998.

[8] R. L. Cruz. "Sched+: Efficient Management of Quality of Service Guarantees". In *Proceedings of the IEEE INFOCOM San Francisco, March 1998*, April 1998.

[9] P. Goyal, H. Vin, and H. Chen. "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks". In *Proceedings IEEE SIGCOMM'96*, August 1996.

[10] M. B. Jones, D. Roşu, and M.-Cătălin Roşu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, Saint-Malo, France, October 5-8 1997.

[11] S. Leffler, M. McKusick, M. Karels, and J. Quaterman. *The Design and Implementation of the 4.4BSD UNIX Operating System*. Addison-Wesley Pub. Co., Reading, MA, 1996.

[12] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[13] Jon C. R. Ben nett and Hui Zhang. "WF$^2$Q: Worst-case Fair Weighted Fair Queueing". In *Proceedings of the IEEE INFOCOM San Francisco, March 1996*, pages 120–128, March 1996.

[14] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks–the single node case. *IEEE/ACM Transactions on Networking*, pages 344–357, June 1993.

[15] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, July 1990.

[16] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[17] I. Stoica and et.al. "A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems". In *Proceedings of IEEE Real-Time Systems Symposium*, December 1996.

[18] F. Toutain. "Decoupled Generalized Processor Sharing: A Fair Queuing Principle for Adaptive Multimedia Applications". In *Proceedings of the IEEE INFOCOM San Francisco, March 1998*, April 1998.