Quality of Service Support for Legacy Applications

José Brustoloni, Eran Gabber, Abraham Silberschatz and Amit Singh
Information Sciences Research Center
Lucent Technologies — Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{jcb, eran, avi, amitsingh}@research.bell-labs.com

Abstract

Many existing digital audio and video applications assume overprovisioning. Such applications perform well on dedicated, lightly loaded systems, but may perform badly in cases of server hot spots or network congestion. We present Eclipse/BSD's solution for running such applications automatically under resource reservations that guarantee the required performance. The key idea in our solution is to give files a new attribute — the resource requirement. *We then interpose between legacy applications and the operating system a modified version of the* libc *library, dynamically linked with applications at load time. The modified library intercepts certain system calls and automatically establishes resource reservations according to the requirements of the accessed files, whether local or remote.*

1 Introduction

Digital audio and video applications require sufficient resources, including CPU, memory, and network and disk bandwidth, in order to perform properly. However, conventional time-sharing operating systems, such as Unix [9] and Windows NT [5], and best-effort networks, such as the current-generation Internet, cannot guarantee resource availability in the amount required by any given application. Therefore, these applications may often not perform as expected.

Discussions on how best to solve this problem are often polarized. One possible solution is to provide *quality of service guarantees* and *admission control*: Operating systems and networks are modified so that the system admits a request only if the system has set aside sufficient resources to guarantee that it will satisfy the request within specified performance bounds. The opposite solution is *overprovisioning*: Simply assume that the total amount of resources tends to be larger than a worst-case workload would demand.

Because deployment of systems with quality of service guarantees has been slow, many existing digital audio and video applications assume overprovisioning. Such applications may perform well on dedicated systems, in absence of other load, but perform poorly, for example, in cases of server hot spots or network congestion.

Slowly but surely, however, systems that do provide quality of service guarantees are being introduced. In a recent paper [3], for example, we described Eclipse/BSD, a new operating system that allows applications to set up *resource reservations* and thereby guarantee performance within certain bounds. We implemented Eclipse/BSD by modifying FreeBSD, a freely available derivative of 4.4 BSD Unix [9].

This paper's contribution is to show how legacy applications may, without being modified, enjoy quality of service guarantees on Eclipse/BSD. This allows certain applications that were written assuming overprovisioning to run well even on systems that are neither dedicated nor lightly loaded. The key idea in our solution is to give files a new attribute — the *resource requirement*. We then interpose between legacy applications and the operating system a modified version of the `libc` library, dynamically linked with applications at load time. The modified library intercepts certain system calls and automatically establishes resource reservations according to the requirements of the accessed files, whether local or remote.

The rest of this paper is organized as follows.

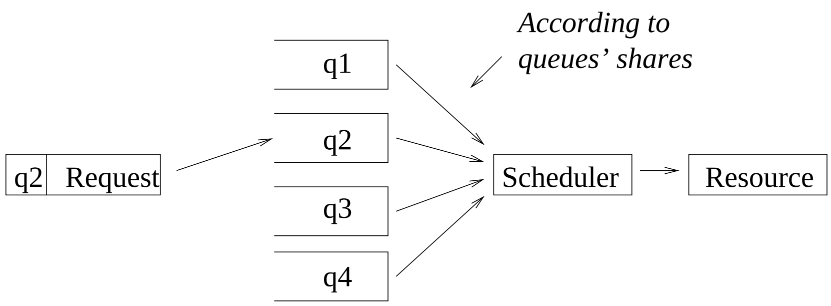

Figure 1: Eclipse/BSD's schedulers apportion resources to each queue according to the queue's share.

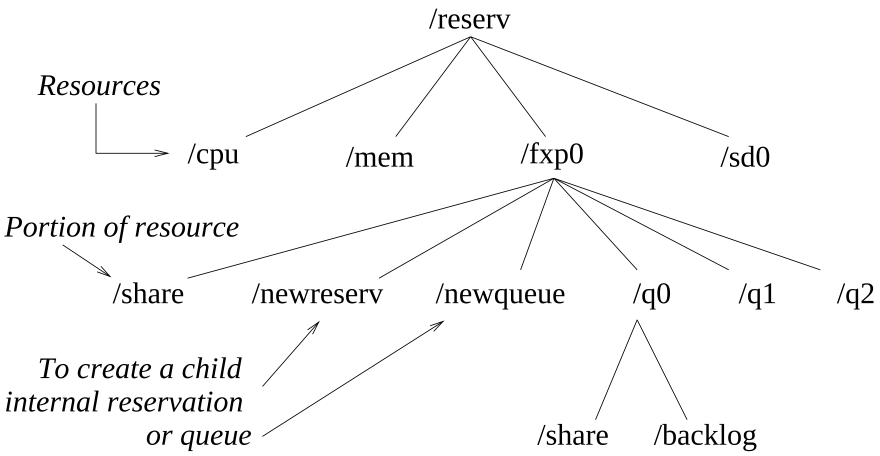

Figure 2: Eclipse/BSD's `/reserv` file system allows applications to create resource reservations.

Section 2 reviews how applications can create resource reservations in Eclipse/BSD. Section 3 then describes how Eclipse/BSD supports resource requirements, including NFS extensions for making ephemeral reservations with the required resources. Section 4 introduces *requirement brokers*, which establish the required resource reservations automatically. The resource reservations created by a broker are children of the application's *root* reservations; Section 5 shows how users can set such reservations from the shell, before invoking the application. Experiments in section 6 demonstrate that brokers can guarantee to unmodified existing applications the required performance. Section 7 discusses related and future work, and section 8 concludes.

2 Resource reservations

This section summarizes our previous work [3], describing how Eclipse/BSD applications can hierarchically create, use, or destroy resource reservations. Eclipse/BSD uses a *hierarchical proportional sharing* scheduler to manage each independent physical resource, such as CPU, memory, network and disk bandwidth. Every request arriving at a scheduler must specify a *queue*, and the scheduler apportions resources to each queue according to the queue's *share*, as shown in Figure 1. Eclipse/BSD's schedulers are different from those of FreeBSD, which do not provide hierarchical proportional sharing. For example, Eclipse/BSD uses, for CPU scheduling, the MTR-LS algorithm [2]; for disk scheduling, the new YFQ algorithm [4]; for network output link scheduling, Bennet and Zhang's H-WF^{2}Q algorithm [1].

Eclipse/BSD applications create resource reservations using the new `/reserv` file system. In `/reserv`, resource reservations are represented by directories. Resource reservations are called *internal reservations* if they can have children, or *queues* if they cannot. Each independently managed physical resource in the system is represented by an internal reservation under `/reserv`: for example, CPU, memory, network and disk bandwidth, as shown in Figure 2. Each resource reservation r has a `share` file, which specifies r's portion of its parent's resources. `share` contains two values: the minimum absolute value, used for admission control, and the weight with which r shares its parent's resources. If r's parent is `/reserv`, then r's `share` file is read-only and represents the resource in its entirety. An application creates an internal reservation or queue as a child of an internal reservation r by opening r's `newreserv` or `newqueue` file, respectively; the open call returns the file descriptor of the newly created `share` file, which is initially null. Writes to `share` files may fail due to admission control. A new command to the `fcntl` system call, F_SHARE_WAIT, allows a process to block until the previous failed write to `share` may succeed if retried.

A process p's *reservation domain* is the list of p's internal *root reservations* for each resource[1]. A process p can open `newreserv` or `newqueue` only in internal reservations that are equal to or descend from one of p's root reservations, and can write into `share` files only in resource reservations that descend from p's root reservations. Queue `q0` of process p's root reservation r is called p's *default queue* for the respective resource. The reservation domain of process `pid` is represented by the read-only file `/proc/pid/rdom`. The reservation domain of processes spawned by process `pid` is initialized to the contents of the writable file `/proc/pid/crdom`, which must contain internal reservations that are equal to or descend from `pid`'s root reservations.

In Eclipse/BSD, different requests may specify the same object but different queues. For example, two processes may be in different reservation domains and each need to use a different disk queue to access a shared file, or a different network output link queue to send packets over a shared socket. Therefore, Eclipse/BSD queues are associated with *references* to shared objects, rather than the shared objects themselves.

In the case of input/output (I/O) objects, such as vnodes and sockets, each *file descriptor* that refers to the object also contains a pointer to a queue. Eclipse/BSD copies that queue pointer to the I/O requests issued on that file descriptor. Note that file descriptors can be private, even if the referred object is not. The file descriptor's queue pointer is initialized to the process's default queue for the respective device: for vnodes, at `open` time; for connected sockets, at `connect` or `accept` time; for unconnected sockets, at `sendto` or `sendmsg` time. However, the `fcntl` system call gets two new commands: F_QUEUE_SET, for setting a file descriptor's pointer to a different queue, and F_QUEUE_GET, for obtaining the name of the queue to which a file descriptor points.

Each resource reservation r has a reference count. The file descriptor of r's `share` file points to r; additionally, `rdom` and `crdom` files and various file descriptors may refer to r, as explained in the previous paragraphs. If r's reference count falls to zero and r's `GC` flag is enabled (default), Eclipse/BSD garbage collects r. Privileged processes can use new commands to the `fcntl` system call, F_COLLECT_SET or F_COLLECT_GET, to set or get a resource reservation's `GC` flag.

3 Resource requirements

The previous section explained why resource *reservations* usually should not be attributes of shared objects, such as files and sockets: Each reference to the object should use its own, possibly different reservation. On the contrary, however, the resource *requirements* of a shared object often are well defined. This is particularly true for media streams: The data rate for digital audio or video strips often is well-known or can be easily bounded. This section describes how Eclipse/BSD supports resource requirements and how new applications may use such support.

In Eclipse/BSD, files get a new attribute, `resource_req`, that specifies the file's nominal data rate, in Kbps. For example, a file containing a video strip might have `resource_req` equal to 1500 (that is, 1.5 Mbps).

Eclipse/BSD stores `resource_req` in the file's inode. Applications use two new commands to the `fcntl` system call, F_RREQ_SET and F_RREQ_GET, to set or get a file's `resource_req`, respectively. The permissions necessary for these commands are the same as those for writing or reading the file, respectively. A new utility, `chrreq`, uses these commands to make it possible to set or get a file's `resource_req` from the shell.

For distributed file systems to be able to set or get `resource_req`, the client/server protocol used may

[1]Note that our current concept of reservation domain is somewhat different from that in our previous work [2].

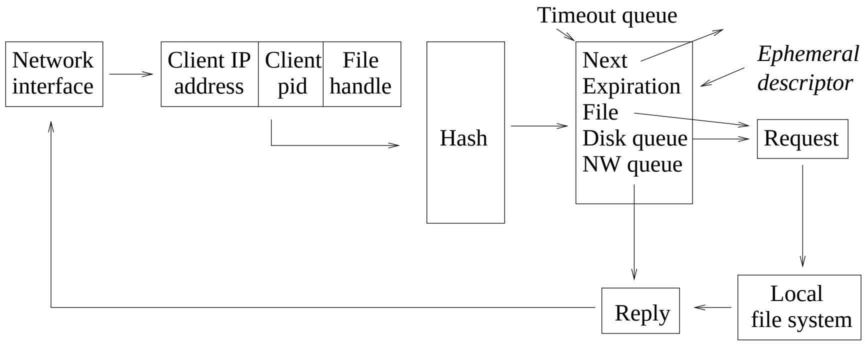

Figure 3: Eclipse/BSD's NFS server uses *ephemeral descriptors* to keep track of queues created for accessing and transmitting files with the required resources.

need to extended, either by introducing new request types or by adding `resource_req` to the file attributes that are transfered in existing request types. For example, in NFS [9], the client could invoke either new procedures or new versions of the existing `GETATTR` and `SETATTR` procedures (Eclipse/BSD opted for the latter[2]).

Eclipse/BSD offers a new command for the `fcntl` system call, F_QUEUE_CREATE_AND_SET, that, applied to a file descriptor *fd* that references a local file f, (1) automatically creates a new disk queue whose `share` matches f's requirements, and (2) sets *fd*'s queue pointer to that queue. The command's argument specifies whether the file will be read and/or written. If f's `resource_req` is null, no queue is created and an error is returned; otherwise, the new queue is created as a child of the process's root reservation for the device where the file is located. If the queue's `share` cannot be set according to `resource_req`, due to admission control, then `share` is closed (causing the queue to be garbage collected) and an error is returned. Otherwise, *fd*'s queue pointer is set to the new queue, `share` is closed, and success is returned. Because *fd*'s is the only remaining reference to the new queue, that queue will be automatically garbage collected when the application closes *fd*.

If, on the other hand, file descriptor *fd* references a *remote* file f, then F_QUEUE_CREATE_AND_SET requires queues that match f's requirements to be created at the server's disk and client and server network interfaces. Client queues and, for stateful distributed file systems, also server queues, can be created analogously to the local case, described in the previous paragraph. However, stateless distributed file systems, such as NFS, require *ephemeral* server queues, which are garbage collected by timeout, instead of when f is closed. In such cases, F_QUEUE_CREATE_AND_SET sets *fd*'s QCS flag (which by default is not set). If *fd*'s QCS flag is set, Eclipse/BSD uses new versions of NFS's `READ` and `WRITE` procedures[2] for I/O through *fd*. Requests in the new versions include not only the file handle, returned by the server, but also the process identifier of the client process. When an Eclipse/BSD NFS server receives one such request, the server hashes the client IP address, process identifier, and file handle to find an *ephemeral descriptor*, as shown in Figure 3. An ephemeral descriptor contains pointers to the file and to a disk queue and a network interface queue matching the file's requirements. The server uses the descriptor's queues on accesses to the file by the respective client. Ephemeral descriptors also contain an expiration time and are linked in a timeout queue. Each time a request hashes to an ephemeral descriptor, that descriptor's expiration is extended. When current time exceeds a descriptor's expiration time, the descriptor's referenced queues and the descriptor itself are garbage collected. On the other hand, if a given request does not hash to any existing descriptor, the server allocates a new descriptor, creates new disk and network interface queues matching the accessed file's requirements, and makes the new descriptor point to the file and the new queues.

Eclipse/BSD does not make F_QUEUE_CREATE_AND_SET implicit in *every* `open` call because, in some cases, a file's resource requirements are irrelevant. For example, when a file is being copied, it does not require any particular data rate, even if the file contains a video strip.

[2] If an Eclipse/BSD NFS client receives an error indication from a legacy NFS server, the client reverts to the standard versions of those procedures.

4 Requirement brokers

The previous section shows that after opening a local or remote file, an Eclipse/BSD application can use a new `fcntl` command, F_QUEUE_CREATE_AND_SET, to guarantee the required resources for accessing the file. Because that command is new, however, it does not benefit legacy applications. This section explains how Eclipse/BSD can support also such applications.

In Eclipse/BSD, users may interpose a *requirement broker* between a legacy application and the operating system. Such broker intercepts certain system calls and automatically establishes resource reservations according to resource requirements.

Requirement brokers that are used by many applications may be implemented by modifying the system's `libc` library, which is dynamically linked with applications at load time. Users may set environment variables to enable or disable brokers implemented within `libc`. On the other hand, a requirement broker for a specific application `/path/app` may be implemented by redefining intercepted system calls in a new library, `/pathl/libapp.so`, that is dynamically pre-linked with `/path/app` before `libc`. A user may cause such pre-linking by moving `/path/app` to `/path/app.leg` and defining a new script `/path/app`:

```
#! /bin/sh
LD_PRELOAD=/pathl/libapp.so /path/app.leg $*
```

Eclipse/BSD's *file system* broker is implemented by modifying `libc` and is enabled by environment variable AUTO_FILE_RESERV. Such broker intercepts only `open` calls; if the call is successful, the broker invokes `fcntl` F_QUEUE_CREATE_AND_SET. The file system broker therefore causes even unmodified existing applications to access local or remote files automatically with the required reservations.

A *network* broker can also be defined for each application-level client/server protocol (e.g., HTTP). Network brokers typically are implemented by specific pre-linked libraries and usually must intercept more system calls than do file system brokers. Network brokers may need to intercept `socket, connect, accept,` and `close` system calls so as to maintain a list of the process's open sockets and their state. They may also need to intercept `write, send, sendto, sendmsg, read, recv, recvfrom,` and `rcvmsg` calls on sockets in order to eavesdrop on requests and replies and keep track of what files are being accessed over what sockets. If the protocol supports only whole-file transfers, file access begins with a request and ends with the corresponding reply. Otherwise, in stateful protocols, access to a file may begin with an open request and end with a close request; in stateless protocols, access to a file may begin with a fetch or store request and may be presumed to have ended a certain timeout after the last fetch or store request.

When network access to a file begins, the network broker consults the file's resource requirements and creates a queue of matching `share` as a child of the process's root reservation for the respective network interface (the broker may invoke `fcntl` F_SHARE_WAIT to block until the required `share` is obtained). The broker sets to the new queue the queue pointer of the file descriptor *fd* used to send or receive the file. The broker also closes `share`, so that *fd*'s is the only reference to the new queue. When access to the file ends, the broker closes *fd*, causing the queue to be garbage collected.

If multiple files are accessed over the same socket, the network broker may need to `dup` the socket's file descriptor for each file. Each duped descriptor can point to a queue with the respective file's required resources. The broker then converts between original and duped descriptors on each fetch or store request or reply on a file.

However, network brokers need not be complex. The network broker for `ftpd` (Eclipse/BSD's FTP server daemon), for example, intercepts the `read`, `open`, `connect`, and `close` system calls. The broker snoops `read` calls on the control socket to detect `RETR filename` (retrieve) and `STOR filename` (store) commands from the client. In such cases, the broker records the file name and corresponding resource requirements. At the subsequent `open` and `connect` calls, respectively, the broker creates and sets disk and network queues matching the requirements. The broker destroys those queues at the subsequent `close` call.

The above file system and network brokers ignore CPU and memory requirements. More sophisticated brokers adjust the application's CPU and memory reservations according to *CPU and memory requirement templates*, which can be *static* or *dynamic*. A static template gives for each application the required CPU (in SPECint95) and memory (in KB), regardless of file accesses. A dynamic template gives, for each application and name extension of a file accessed by the application, the additional CPU and memory requirement. Additional requirements are the sum of a fixed term and the product of a variable term and the accessed file's `resource_req`. CPU and memory requirement templates are experimentally determined and maintained by the system administrator.

5 Setting an application's reservation domain from the shell

As explained in the previous section, requirement brokers act on behalf of legacy applications to automatically create queues that match the resource requirements of accessed files. Such queues are children of the application's root reservations. This section shows how users can set an application's reservation domain (i.e., the root reservations) from the shell.

Eclipse/BSD provides a new utility, `newreserv`, for creating a new internal reservation that is a child of a named internal reservation. The created internal reservation is not garbage collected when the utility exits because the utility has set-user-id mode set and runs as a privileged process, which can use `fcntl` F_COLLECT_SET to prevent a resource reservation's garbage collection. `newreserv` prints the name of the new internal reservation.

Before using `newreserv`, users will typically need to invoke `ps` to determine the shell's process id, s, and then find the shell's reservation domain by invoking `cat /proc/`s`/rdom`. Given the shell's root reservations, users can then create new internal reservations using `newreserv`. To set the share of resource reservation b of resource a, users may invoke `cat > /reserv/`a`/`b`/share`. Finally, users can use `cat > /proc/`s`/crdom` to set the shell's `crdom` file to a list of internal reservations that are equal to or descend from the shell's root reservations. Applications will be started from the shell with reservation domain (`rdom`) equal to the shell's `crdom` file.

6 Experimental results

This section demonstrates experimentally that Eclipse/BSD's brokers work as expected both in local and in remote file accesses, automatically reserving resources according to the file requirements.

We performed our experiments on a pair of PCs connected by a lightly loaded Ethernet at 100 Mbps. Both PCs ran either the FreeBSD 2.2.8 or the Eclipse/BSD operating system. The server PC had a 266 MHz Pentium II CPU, 64 MB RAM, and a 9 GB SCSI disk. The client PC had a 133 MHz Pentium CPU, 32 MB RAM, and a 4.3 GB IDE disk.

A series of experiments tested Eclipse/BSD's NFS extensions and file system broker. In these experiments, applications A, B, C, and D simultaneously read each a different 100 MB file (respectively, f_A, f_B, f_C, and f_D) with requirement respectively of 2.4, 1.8, 1.2, and 0.6 Mbps. Applications ran either all on the client PC or all on the server PC; files remained always on the disk of the server PC. To prevent cache effects, applications read very large unrelated files before each experiment. There was no other load on the PCs. To ensure repeatable and nearly worst-case conditions, sectors of each file were perfectly interleaved on the disk. Each application measured its throughput by, each two seconds, dividing by such period the number of bytes read by the application since the previous measurement.

Figures 4 and 5 show the throughput obtained by each application when the applications ran on the server PC (local file access) using either FreeBSD or Eclipse/BSD and file system broker, respectively. The figures show that FreeBSD has much higher throughput variance and that Eclipse/BSD's file system broker automatically and correctly discriminates the different file requirements[3].

Similar results are shown in Figures 6 and 7 for the case where the applications ran on the client PC (remote file access). FreeBSD does not discriminate the

[3]Because in these experiments the files are perfectly interleaved but are sequentially read at different rates, seek overheads gradually increase. Our current implementation uses a disk scheduling algorithm, YFQ [4], that does not properly account for the absolute impact of seek overheads (it guarantees only proportional sharing of net disk bandwidth). This causes the decaying absolute throughputs shown in Figures 5 and 7. We verified experimentally that absolute throughputs are nearly proportionally shared *and* constant if files are accessed randomly (causing average seek overheads to remain nearly constant).

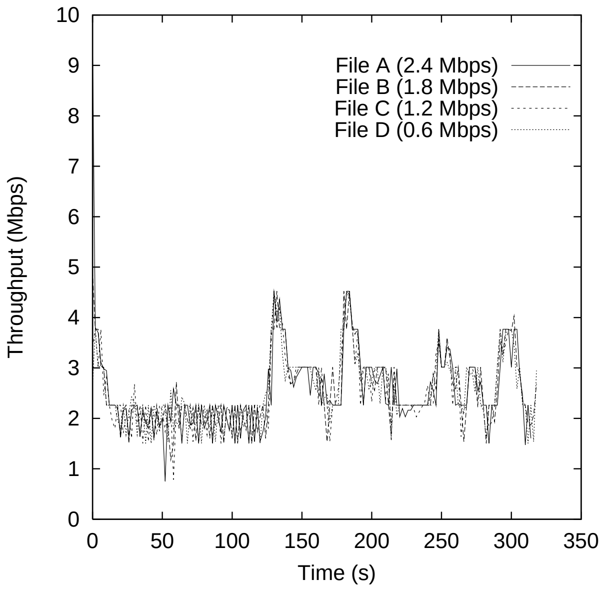

Figure 4: FreeBSD gives widely variable but similar throughput to all applications accessing local files.

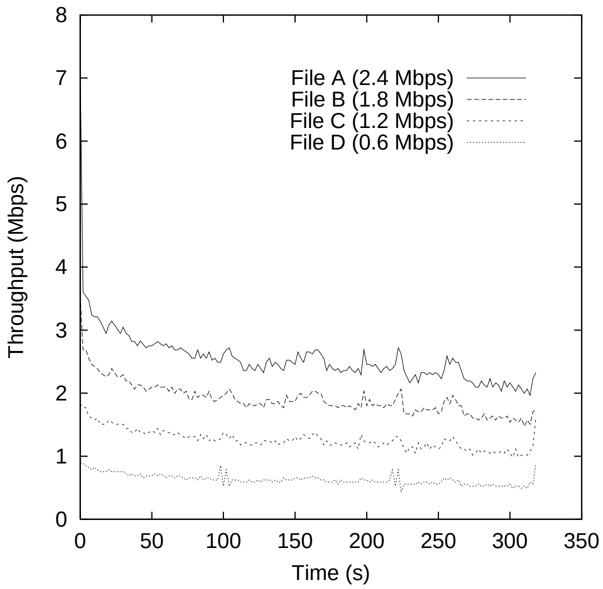

Figure 5: Eclipse/BSD's file system broker automatically creates disk queues that give each application the required performance for accessing local files.

applications, giving too little or too much bandwidth to different applications at different times. On the contrary, Eclipse/BSD's file system broker automatically gives to each application the required performance.

A final experiment tested Eclipse/BSD's FTP broker. In this experiment, each of three FTP clients A, B, and C simultaneously retrieved a large (> 100 MB) file (respectively, f_A, f_B, and f_C) with a requirement respectively of 3.6, 2.4, and 1.2 Mbps. Our measurements showed that Eclipse/BSD's FTP broker indeed caused each file to be transferred with the required bandwidth.

7 Related and future work

An alternative that may be of interest is associating `resource_req` with file *type*, instead of with individual files, as we did here. For example, the system could use a database that specifies that all files whose name has a certain extension have a corresponding `resource_req`. Such scheme would be less flexible than the one presented here, since presumably only the system administrator would be able to change the database. On the other hand, a replicated database could make consulting a file's requirements faster, particularly in wide area networks.

SLIC [6] allows interposition of kernel events, such as system calls and signals, by trusted kernel extensions. Requirement brokers could be implemented as SLIC extensions, but then their installation would require intervention of the system administrator, unlike the solution presented here. Requirement brokers actually need not be trusted, since they use system primitives just like any application. This allows us to implement brokers at user level, as described in this paper, by modifying `libc`.

Odyssey [11] is a framework for resource management in mobile systems. Odyssey does not provide quality of service guarantees, which are difficult to achieve in mobile environments. Instead, Odyssey monitors available resources and notifies applications when relevant changes happen; applications then adapt by operating at different fidelity levels. Odyssey's Cellophane is analogous to a network broker: It permits a legacy application, Netscape, to adapt using Odyssey primitives. Interposition is in this case simplified by Netscape's *proxy* facility, which redirects requests and makes it unnecessary to eavesdrop on sockets.

Eclipse/BSD provides hierarchical proportional re-

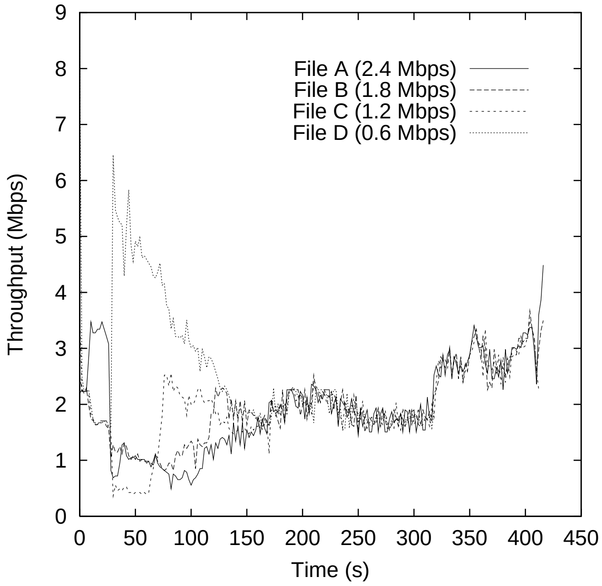

Figure 6: FreeBSD's NFS does not discriminate between applications, giving bandwidth that is sometimes excessive, sometimes deficient.

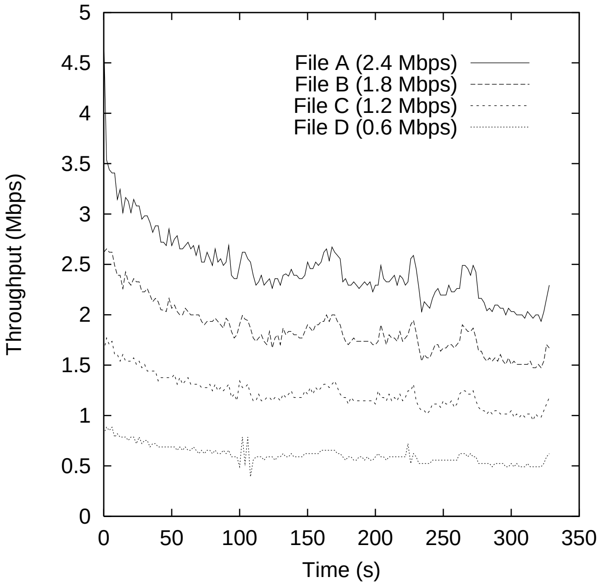

Figure 7: Eclipse/BSD's file system broker and NFS extensions give to each application the required performance.

source sharing, which is useful in soft real time and certain multimedia applications. It would be interesting to investigate whether Eclipse/BSD's notions of resource requirements and requirement brokers could be adapted to systems that support hard real time applications, such as Nemesis [8], SMART [10], and Rialto [7].

8 Conclusions

Eclipse/BSD is an operating system that is derived from FreeBSD and that provides hierarchical proportional resource sharing. We previously had shown how new applications can exploit Eclipse/BSD's resource reservations to guarantee performance within certain bounds. In this paper, we presented how Eclipse/BSD supports the complementary notion of resource requirements, both in the local and the remote case. In particular, we described extensions to NFS that enable it to make reservations and thereby provide a required quality of service. We then demonstrated experimentally that by interposing requirement brokers — modified versions of `libc` that intercept certain system calls — Eclipse/BSD can run unmodified legacy applications automatically under reservations that guarantee the required performance.

Quality of service support is likely to evolve dramatically in the near future with the introduction of differentiated-service networks and new operating systems. We believe that the notion of resource requirements and the interposition of requirement brokers similar to the ones described here may help existing applications benefit from the improved quality of service of future systems.

Acknowledgments

We thank John Bruno and Banu Özden for valuable discussions during the design phase of this work.

References

[1] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[2] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Do-

mains", in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. "Retrofitting Quality of Service into a Time-Sharing Operating System", to appear in *Proc. Annual Tech. Conf.*, USENIX, June 1999.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Özden and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees", to appear in *Proc. ICMCS'99*, IEEE, June 1999.

[5] H. Custer. "Inside Windows NT", Microsoft Press, 1993.

[6] D. Ghormley, D. Petrou, S. Rodrigues and T. Anderson. "SLIC: An Extensibility System for Commodity Operating Systems", in *Proc. Annual Tech. Conf.*, USENIX, June 1998.

[7] M. Jones, D. Rosu and M. Rosu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 198-211.

[8] 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 *JSAC*, 14(7), IEEE, Sept. 1996, pp. 1280-1297.

[9] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley Pub. Co., Reading, MA, 1996.

[10] J. Nieh and M. Lam. "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 184-197.

[11] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker. "Agile Application-Aware Adaptation for Mobility", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 276-287.