Better operating system features for faster network servers

Gaurav Banga   Peter Druschel
Dept. of Computer Science
Rice University, Houston, TX 77005
{*gaurav,druschel*}*@cs.rice.edu*

Jeffrey C. Mogul
Western Research Laboratory
Compaq Computer Corp., Palo Alto, CA, 94301
mogul@pa.dec.com

Abstract

Widely-used operating systems provide inadequate support for large-scale Internet server applications. Their algorithms and interfaces fail to efficiently support either event-driven or multi-threaded servers. They provide poor control over the scheduling and management of machine resources, making it difficult to provide robust and controlled service. We propose new UNIX interfaces to improve scalability, and to provide fine-grained scheduling and resource management.

1 Introduction

The performance of Internet server applications on a general purpose operating system is often dismayingly lower than what one would expect from the underlying hardware. Internet servers also suffer from other undesirable properties such as poor scalability, unfair resource allocation, susceptibility to livelock under excess load, instability under denial of service attacks, and inability to prioritize handling of requests.

The cause of these problems is a fundamental mismatch between the original design assumptions of existing operating systems and the requirements of modern server applications. Current general purpose operating systems were designed for efficient timesharing. In such environments, processes spend most of their time in user mode, infrequently invoking the kernel to access slow I/O devices. In contrast, an Internet server application often manages huge numbers of simultaneous network streams, with unpredictable event arrivals. The application makes frequent system calls, spending significant time executing in the kernel.

Many features of modern operating systems were designed without consideration of scaling to large sets of resources; $\mathcal{O}(N)$ behavior that was acceptable with $N = 20$ is problematic when $N = 10000$. This is especially true for system calls used for event management, such as select(). In addition, scheduling and resource management does not extend to the execution of significant parts of kernel code. The application has no control over the consumption of many system resources, such as kernel memory that the kernel manages on behalf of the application. This makes it difficult or impossible to prevent low-priority clients from hogging resources.

Researchers have been aware of these problems for several years [20, 23], and together with system vendors have devoted much effort to improving Internet server performance. Some of this has been as simple as tuning kernel parameters; in other cases it has been necessary to improve the implementation of some kernel features, such as the protocol control block (PCB) lookup algorithm in BSD-based systems [21, 30], and the select() system call [6].

Application writers have also worked to make more efficient use of existing operating system services. While early servers used a process-per-connection approach, recent servers [10, 31, 35] use a single-process event-driven architecture, to reduce context-switching overhead. Even these servers have some scalability problems [6].

The work cited above has generally assumed the use of the existing system-call interface, which limits the degree to which performance problems can be addressed. For instance, the scalability of UNIX-based event-driven servers is limited by the inherently linear-time select() system call [6].

We propose extending the UNIX system call interface to provide more efficient support for Internet server applications. We discuss two control models for servers (event-driven and multi-threaded), and examine what they require from the operating system. We look at how such applications need to control scheduling and kernel resource consumption. We then describe new application programming interfaces (APIs) to support these requirements. This paper does not address issues related to efficient operating system support for data movement. Other research has addressed those issues [2, 28, 32].

2 Evolution of server execution models

We begin by describing the evolution of Internet server execution models. To be concrete, we focus on Web servers. However, most of the issues we discuss apply to other servers such as proxy, mail, and file servers.

The earliest Web servers forked a new process to handle each HTTP connection, following the classical UNIX model. The forking overhead quickly became a problem, and subsequent servers (such as the NCSA httpd [26]) used a set of pre-forked processes. In this model, a master process accepts new connections and passes them to pre-forked slave processes to be handled. In newer process-per-connection servers, e.g. Apache [4], there is no master process. Instead, each server process calls accept() directly to accept new connections.

Multi-process servers can suffer from large context-switching overhead, so many recent servers use a single-process event-driven architecture. (Event-driven servers designed for multiprocessors use one thread or process per processor.) An event-driven server uses the select() system call to simultaneously wait for events on all connections being handled by the server. When select() delivers one or more events, the server's main loop invokes handlers for each ready connection. Squid [31], Zeus [35] and thttpd [33] use this architecture.

Another alternative is the single-process multi-threaded model, in which each connection is assigned to a dedicated thread. The thread scheduler is responsible for time-sharing the CPU between the various server threads. Since there is only one server process, context-switching overhead is much lower than in a process-per-connection architecture. However, efficient support for huge pools of threads is not always available, and so some servers use a hybrid approach, in which a moderate-size pool of threads is multiplexed among many connections, using events to control the assignment of connections to threads.

The discussion above assumes requests for static documents. HTTP also supports requests for dynamic documents, whose contents depend on the particular parameters and arrival time of the request. Dynamic documents are typically created by auxiliary third-party programs, which run as separate processes. To make the construction of such auxiliary programs easier, several standard interfaces have been defined to govern communication between Web servers and such programs. Examples include CGI [8] and FastCGI [13]. The earlier interface, CGI, creates a new process to handle each dynamic document request. The newer FastCGI allows persistent auxiliary processes. Microsoft and Netscape have also defined new library interfaces [17, 27] to allow the construction of third-party components that can reside in the main server process, when fault isolation is not an issue.

Many Web servers are moving towards an architecture where a small set of processes implement the functionality of the server. There is one main server process, which serves all static documents. Dynamic documents are created by either library code within the main server, or by auxiliary processes whose code needs to be kept apart from the other components of the server for reasons of fault isolation. This has the advantage that the overhead of switching context between protection domains is incurred only if absolutely necessary. However, structuring a server as a small set of processes leads to certain important problems. This is the subject of the next section.

3 Missing operating system support

As we noted in Section 2, current operating systems lack efficient and scalable support for modern servers. In this section, we describe these inadequacies in detail, and discuss their implications for server performance.

3.1 Event-driven servers

The performance of event-driven servers depends critically on three things. First, the event delivery and handling mechanism must be efficient and scalable. Secondly, the single thread of control must never block while handling an event. If an event-handler were to block, this would delay the delivery and handling of subsequent events. Finally, any changes in a server's environment should be reported to the server asynchronously; a server should never have to resort to a status poll on the resources that it manages for correct operation.

A UNIX program can use either signals or the select() (or poll()) system call to wait for events without blocking. The signal mechanism (e.g., using SIGIO to indicate an I/O event) is usually a poor choice, because the signal is delivered without any indication of which descriptor is now ready. Signals therefore do not scale to support many connections.

Although select() scales much better than signals, both the interface and traditional implementation of select() scale poorly with large numbers of descriptors. The implementation can be improved [6], but the interface inherently imposes costs linear in the number of descriptors (and makes optimizing the implementation difficult). The interface scales poorly because it passes information about *all* established connections from user-space to the kernel at each wait-for-next-event request. A similar amount of information is passed from the kernel back to user-space at each event notification. Moreover, the application must then scan a bitmap, whose size is proportional to the number of connections, to discover which of these are ready.

The lack of non-blocking I/O support in current operating systems also limits the performance of event-driven servers. In UNIX, a single page fault or disk read can cause the server process to be suspended for up to hundreds of milliseconds. This prevents any progress, even on unrelated connections that could be handled without additional I/O.

Some system vendors have implemented the POSIX aio interface for non-blocking disk I/O. This allows a server to avoid blocking on explicit disk reads and writes, but it does not avoid other synchronous disk operations (directory lookups, the stat() system call, etc.). The aio interface is not used by most libraries, making it difficult to compose programs from independently-developed components. Perhaps most problematic, though, is that select() cannot be used to detect aio completion events, and the aio interface does not work with sockets. This forces an application to use cumbersome methods to wait simultaneously for network and disk events.

3.2 Multi-threaded servers

Previous research has exposed the limitations of thread support in current operating systems [3]. Kernel threads impose the overhead of a kernel call for each synchronization and context-switch operation. User-level threads have efficient context-switching and synchronization, but if any one thread blocks for I/O, all other threads in the process stall.

To rectify these problems, Anderson et. al. [3] proposed "scheduler activations", which are kernel thread-like entities that provide execution contexts to user-level threads. Context-switching is usually handled at user-level without changing the underlying scheduler activation. When a user-level thread blocks, the kernel provides another activation to allow other unblocked user-level threads to run. Thus context-switching is fast, and a blocked thread does not stall all other threads of the process.

Operating system thread support has other limitations peculiar to Internet servers. A multi-threaded Web server may have hundreds or thousands of open connections. So, the kernel must efficiently support such "massively-threaded" processes. Current thread implementations, whether based on scheduler activations or otherwise, do not scale to such huge numbers of threads. The main challenge is to minimize the context-switching overhead and TLB miss rates that result from maintaining a large number of thread stacks. Excessive synchronization can also be a problem. Because of these reasons some servers, such as the Inktomi Traffic Server, use a hybrid control model [34]. Such servers use a moderate number of threads, each of which executes an event-driven state machine. This gets around some problems of pure event-driven servers, such as blocking I/O system calls, while still keeping thread management overhead low.

Unfortunately, very little has been published about the use and performance of purely multi-threaded servers, even though several important high-performance servers, e.g. *AltaVista*, have successfully adopted this approach. There has been a lot of research in the runtime systems community on improving the performance of massively threaded applications [9, 15] by reducing the storage management overhead. However, these approaches have not yet been applied to general-purpose operating systems. In this paper, we will concentrate on providing support for an event-based control model.

3.3 Scheduling and resource management

Most operating systems treat a process, or a thread within a process, as the schedulable entity. The process is also the "chargeable" entity for the allocation of resources, such as CPU time and memory. The system's scheduling and memory allocation policies attempt to provide fairness among these entities, and graceful behavior of the system under various load conditions.

However, in most operating systems, the kernel generally does not control or properly account for resources consumed during the processing of network traffic. Most systems do protocol processing in the context of software interrupts, whose execution is either charged to the unlucky process running at the time of the interrupt, or to no process at all. Moreover, software interrupts have strictly higher priority than the execution of any user-level code. This can lead to scheduling anomalies, decreased throughput, and starvation or livelock [12, 24]. This is particularly important for servers because they are, by their nature, particularly network intensive.

The LRP network subsystem architecture [12] was designed to address these problems by more closely following the process-centric model. In LRP, resources spent in processing network traffic are associated with and charged to the application process that caused the traffic. Network processing is scheduled at the priority of the corresponding process, and excess incoming traffic is discarded early. LRP systems exhibit better fairness between processes, and provide stable overload behavior.

However, even such a "faithful" implementation of the process-centric resource model fails to support single-process servers. This is because, in a process-centric model, the kernel does not generally distinguish between independent sub-activities within a process. For example, a process cannot specify the relative priorities of its various network connections, or the priority of establishing new connections relative to servicing the existing connections. Thus, even though an application can prioritize the handling of connections in user-level code, this does not go very far in controlling the relative progress of connections. This is because most of the work performed in network processing is done in the kernel, and cannot be controlled by the application.

This lack of control over kernel resources makes it difficult to build a Web server to provide differentiated quality of service (QoS) to its clients [1]. For example, consider a Web server which would like to provide clients with differentiated service depending on a variety of factors, such as the difference in access fees paid by corporate and home-user clients, or the difference in fees paid by the owners of various content provided by the server. Unfortunately, although such situations already abound on the Web, and will become more widespread, servers running on traditional operating systems cannot provide the needed support.

Under overload conditions, the lack of effective resource management limits a server's ability to service existing connections instead of accepting new ones. This leads to poor server throughput and instability under heavy load, making servers susceptible to denial-of-service attacks. For instance, a high rate of connection establishment requests sent to a server can potentially bring it to its knees [7].

4 Operating system support for server applications

In this section, we propose new operating system features to support server applications. We first describe a fine-grained resource management system for servers; this is the key to enabling robust and controlled service, independent of the execution model (thread-based or event-driven).

We then propose improved support for event-driven servers. While the choice between execution models remains complex, we believe that the performance distinction between thread-based and event-driven servers is probably secondary to many other design and implementation decisions, such as the order in which events are handled, or need for multiprocessor support. Practical constraints may favor a hybrid model, with a moderate number of threads and an event-based notification mechanism.

As mentioned in Section 3.2, the issues involved in efficiently supporting threads are relatively well understood. Thus, we concentrate here on efficient support for event-driven servers.

4.1 Resource containers

We propose a new model for fine-grained resource management and scheduling. This model is based on a new operating system abstraction called a *resource container* [5], which encompasses all system resources that the server uses to service a particular client connection. All kernel processing for a connection is charged to the appropriate container, and scheduled at the priority of the container.

Each resource container has a priority value that is used to control the scheduling of any threads that are associated with the container. A container's priority also controls the allocation of other resources, such as kernel memory. The kernel carefully accounts for the CPU and memory resources it uses for a container. Application processes can access this resource usage information, and use it to adjust the priority of the container.

The notion of a resource container is a generalization of the process model in current operating systems, where the process itself is the resource container. (This is only approximately true in vanilla UNIX, where kernel resource utilization is often charged to the wrong process, or to none at all; it is more nearly true in a LRP [12] system.) In current systems, a process has a dual function: it serves as a protection domain, and as a resource container. As a protection domain, a process provides a mechanism for isolation between applications. As a resource container, a process provides the operating system with resource principals between which the system's resources are to be shared. Unfortunately, this coincidence between protection domains and resource principals is not always appropriate, as discussed below.

Usually an application consists of a single process, and performs a single task. For such applications, the desired unit of isolation and resource consumption is identical, and the process abstraction suffices. Figure 1 shows an application, which uses a single process to perform a single I/O-intensive coherent task. The kernel resource consumption of such applications is largely uncontrolled. LRP extends a process's resource container into the kernel leading to the situation shown in Figure 2. Even in LRP, the resource container is still firmly linked with the process abstraction.

Sometimes an application, performing a single coherent task, is split up into multiple protection domains. A reason for doing this, for example, is the need to provide fault isolation between different components of the application which might come from different vendors. For such applications, the desired unit of protection (the process) is different from the desired unit of resource management (all the processes of the application). A multi-process application trying to perform a single, mostly user-mode, task is shown in Figure 3.

In yet another scenario, an application consists of a single process, and tries to accomplish multiple independent tasks. Such applications use a single protection domain in order to minimize context-switching overhead. For such applications, the correct unit of resource management is smaller than a process: it is the set of all resources being used by the process to accomplish a single task. Single-process servers are of this type. Figure 4 shows a single-process multi-threaded Internet server.

In realistic single-process Internet server systems, the situation is a combination of the last two scenarios. Usually, a single process manages all connections. Sometimes, however, the situation changes to a multi-process scenario. This happens, for example, when the main server process forks off a CGI script to handle a dynamic document request. In this case again, the desired unit of re-

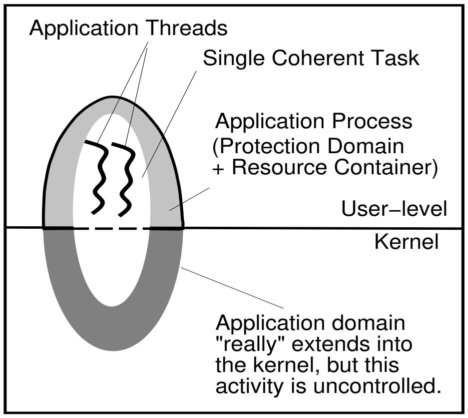

Figure 1: An I/O-intensive application.

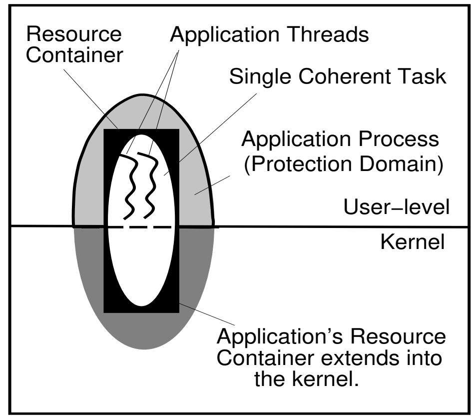

Figure 2: An I/O-intensive application in LRP.

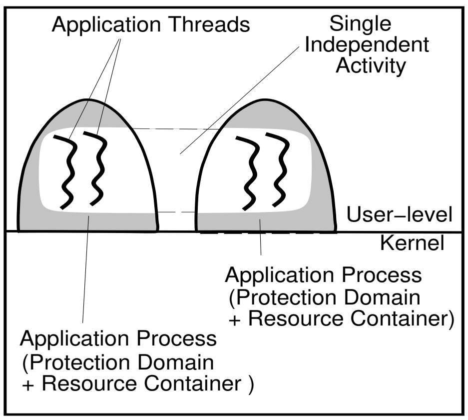

Figure 3: A multi-process application.

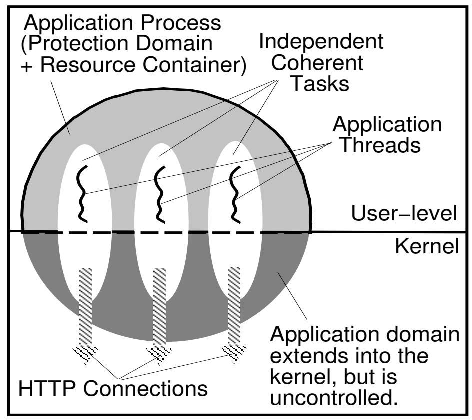

Figure 4: A multi-threaded Web server.

source management differs from a process. This breakdown of equivalence between a protection domain and a resource container in server systems provides the motivation to develop an explicit resource container abstraction.

In summary, the coincidence of "protection domain" and "resource container" in the process abstraction is appropriate for classical programs, because they seldom need to control the rate of progress of distinct activities within themselves. Even when this is important, it can be accomplished using user-level mechanisms, because kernel resource utilization forms an insignificant part of the total resource usage of the process. This is not true for a kernel-intensive server that needs to ensure the controlled, potentially prioritized progress of its various independent connections.

The implementation of resource containers in UNIX involves several changes to the process and thread mechanisms, and to the kernel execution model. Specifically, new system calls are needed to allow an application to create a new container, and to create various types of association between threads and containers. The kernel execution model in the new system is a generalization of the LRP approach [12].

Resource containers are named within a process by file descriptors, allowing the use of several existing system calls to manipulate resource containers. For example, `sendmsg()` can be used to transfer descriptors, and hence containers, between protection domains.

Once the kernel explicitly supports resource containers, server applications can use them to provide robust and controlled behavior. Consider first a single-process multi-threaded Web server that uses a dedicated thread to handle each HTTP connection. In the new model, this type of server will create a container for each new connection. It will then pick a thread from its pool of free threads to service this container. This is shown in Figure 5.

Subsequently, any processing for a connection will be charged to the associated container. If a particular connection (for example, a long file transfer to a well-connected host) is consuming a lot of system resources, this would be reflected in high resource usage values for this connection's container. The scheduling priority of the associated thread will

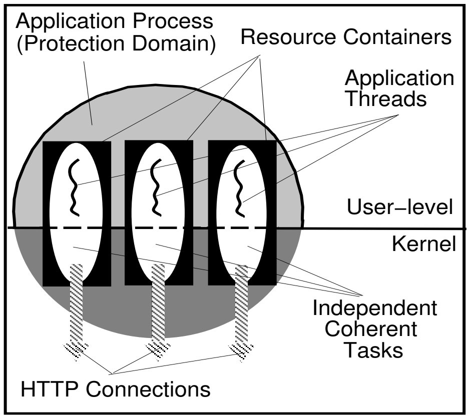

Figure 5: Resource containers in a multi-threaded server.

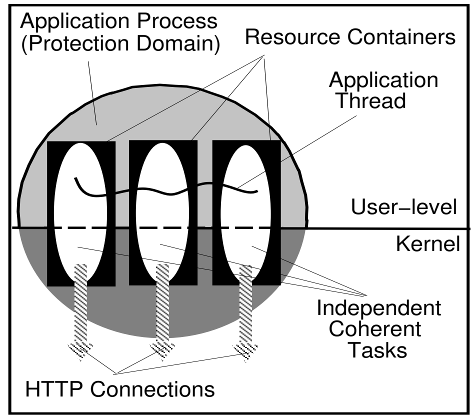

Figure 6: Resource containers in an event-driven server.

decay and the system will preferentially schedule threads handling other connections.

Consider next an event-driven server running on a uniprocessor. In the new resource management model, such a server will create a container for each new connection. The kernel will compute the scheduling priority of the single thread of this server based on the resource consumption of all the containers in the server process. When the server does processing for some connection, it will dynamically charge the resources being consumed to the container corresponding to this connection. Figure 6 depicts this situation.

If a connection consumes a lot of resources, this manifests itself as higher resource usage values for the corresponding container, available to the application via a new API. The server can then use this information to adjust the internal priority value for this container and control the resources that are subsequently expended for this container. A server based on user-level threads may employ containers in a similar manner.

In both kind of servers, when a CGI request comes along, the connection's container is passed to the process which creates the dynamic document. If traditional CGI is being used, this container is simply inherited by the CGI process. With persistent CGI, the connection's container is passed to the CGI process along with the CGI request.

Resource containers also enable the assignment of different priorities to incoming requests for connections from different sources. To support this, we define a new `sockaddr` namespace that includes, besides the local port number and Internet address, a filter specifying a set of foreign addresses. The filter is specified as a tuple consisting of a template address and a CIDR network mask [29]. The server application may then use the `bind()` system call to bind multiple server sockets, each with the same <*local-address, local-port*> tuple but with a different <*template-address, CIDR-mask*> filter. Each filter assigns all requests from a particular client, or set of clients, to one of these sockets. By associating a different container with each such socket, the server application can assign different priorities to different sets of clients, prior to listening for and accepting new connections on these sockets.

This ability to differentiate between incoming requests from different sources greatly increases the ability of a server to provide prioritized handling of clients. For example, a proxy server at an Internet service provider can use this to control the priorities at which different classes of clients are handled. Placing listen sockets into resource containers also allows a server to control the priority of accepting new connections relative to servicing the existing ones. In particular, by creating a listen socket with a priority of zero, whose socket filter is bound to a particular client, a server can protect itself from a denial-of-service attack from a malicious client.

4.2 Efficient event support

To improve the efficiency of event-based servers, we propose two new APIs. One tells the kernel the file and socket descriptors on which an application is waiting for events. The other provides event notifications to the application, preserving the application's priority assignments. The `select()` system call merges both functions; splitting them into separate calls increases flexibility while avoiding the inherent unscalability of `select()`.

Once an application becomes interested in events on a descriptor, it may remain interested in this descriptor for a lengthy period. Our first API allows the application to inform the kernel when this period begins and ends, rather than (as in `select()`) passing this information repeatedly. In effect, the kernel maintains an INTERESTED set for each thread, persisting across many system calls.

The `declare_interest()` system call asserts an application's interest in events on a set of descriptors. The revoke_interest() call indicates that it is no longer interested in events on a set of descriptors. For example, when a server acccepts a connection, it calls declare_interest() with the new socket as an input. Similarly, when a proxy cache starts an asynchronous I/O on a disk file, it calls declare_interest() with the disk file descriptor as input.

When an event (e.g., a received packet or completed disk read) arrives for a descriptor in the INTERESTED set of a thread, the kernel adds the descriptor to a SIGNALLED_EVENTS set it maintains for the thread. Our second API, the dequeue_next_events() system call, allows the application to obtain this set of descriptors with pending events. The application may either ask for the entire set, or for a limited number of descriptors. This allows a multiprocessor application to distribute event processing across the CPUs: each thread can ask for just one descriptor, leaving the rest for other CPUs. Alternatively, a thread can request multiple descriptors, amortizing the cost of this system call across several events.

The resource container mechanism allows the application to assign priorities to descriptors. The kernel delivers descriptors, via dequeue_next_events(), in priority order, allowing the application to entirely postpone its processing of low-priority events. The kernel could use an efficient data structure, such as a priority queue, to represent the SIGNALLED_EVENTS set.

5 Related Work

Resource containers are similar to a number of mechanisms that have been developed recently to support fine-grained resource management. We will discuss these mechanisms, and their relationship to resource containers, in some detail below.

The Scout operating system [25] is based on the *path* abstraction, representing an I/O channel (such as a TCP connection) through a multi-layered system. A path encapsulates the specific attributes of an I/O channel, and allows access to these attributes across layers. Resource containers, in contrast to paths, allow the application to treat the resources consumed by several channels as being part of the same activity. Moreover, the composition of a path is limited by the router graph specified at kernel-build time; resource containers encompass arbitrary sets of resources at run-time. Finally, paths imply a fundamental redesign of the entire operating system.

The exokernel approach [19] gives applications as much control as possible over raw system resources. Functions implemented by classical kernels are instead provided in user-mode libraries. In a network server built using an exokernel, the application controls essentially all of the protocol stack, including the device drivers; the storage system is similarly exposed. It seems feasible to implement the resource container abstraction in an exokernel library operating system, since the exokernel delegates most resource management to user code.

At a superficial level, the functionality provided by resource containers is similar to that provided by a number of abstractions developed in the context of multimedia and real-time operating systems. These include the processor capacity reserves of Mercer et. al. [22], the *activities* [18] of Rialto, the *migrating threads* of Mach [14] and AlphaOS [11], and the *shuttles* of Spring [16]. The chief differences are related to the more general nature of resource containers, and the fact that none of the above abstractions addresses the problem of controlling resource usage of kernel I/O processing.

Almeida et al. [1] attempted to implement QoS support in a modified Apache [4] Web server, running on the Linux operating system. QoS requirements were mapped onto process priorities. The authors were able to provide differentiated HTTP service to different QoS classes. However, the effectiveness of this technique was limited by their inability to control kernel-mode resource consumption, or to differentiate between existing connections and new connection requests. Also, this approach does not extend to event-driven servers.

6 Summary

We discussed the need for operating system support for Internet servers, and identified two dimensions of this support. First, the operating system needs to allow a process to manage the kernel's consumption of resources for individual connections. We proposed the *resource container*, a kernel-supported abstraction encompassing the resources associated with a connection. The process may assign a priority to each resource container, allowing the kernel to favor high-priority connections.

We also addressed the lack of efficient support for an event-driven server execution model. We proposed a new API for event handling, which should scale to large numbers of connections per process.

Acknowledgments

We are grateful to Deborah Wallach and Carl Waldspurger for their comments on earlier drafts of this paper, and for their help in proofreading.

References

[1] J. Almeida et al. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proc. of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[2] E. W. Anderson and J. Pasquale. The Performance of the Container Shipping I/O System. In *Proc. of the 15th ACM Symp. on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[3] T. E. Anderson et al. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proc. of 13th ACM Symp. on Operating Systems Principles*, October 1991.

[4] Apache. http://www.apache.org/.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. *Submitted for publication*, http://www.cs.rice.edu/~gaurav/papers/submit98.ps, Aug. 1998.

[6] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[7] S. M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *Computer Communication Review*, 19(2):32–48, Apr. 1989.

[8] The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[9] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proc. of the ACM Symp. on Prog. Lang. Design and Impl.*, Philadelphia, PA, May 1996.

[10] A. Chankhunthod et al. A Hierarchical Internet Object Cache. In *Proc. of the USENIX Annual Technical Conference*, Jan. 1996.

[11] R. K. Clark et al. An architectural overview of the alpha real-time distributed kernel. In *Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, WA, Apr. 1992.

[12] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[13] Open Market. FastCGI Specification. http://www.fastcgi.com/.

[14] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. of the Winter USENIX Conf.*, Jan. 1994.

[15] D. Grunwald and R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. In *Proc. of the* 2nd Intl. Conf. on Arch. Support for Prog. Lang. and Operating Systems, Cambridge, MA, Oct. 1996.

[16] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. of the Summer USENIX Conf.*, Cincinatti, OH, June 1993.

[17] Microsoft Corp. ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[18] M. B. Jones et al. Modular real-time resource management in the Rialto operating system. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995.

[19] M. F. Kaashoek et al. Application performance and flexibility on Exokernel systems. In *Proc. of the Sixteenth Symposium on Operating System Principles*, Saint-Malo, France, Oct. 1997.

[20] R. E. McGrath. Performance of Several HTTP Demons on an HP 735 Workstation. http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html, Apr. 1995.

[21] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming tcp packets. In *Proc. of the SIGCOMM '92 Conference*, Aug. 1992.

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

[23] J. C. Mogul. Operating system support for busy internet servers. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995.

[24] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[25] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[26] NCSA httpd. http://hoohoo.ncsa.uiuc.edu/.

[27] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. Technical Report TR97-294, Rice University, CS Dept., Houston, TX, 1997.

[29] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, Sept. 1993.

[30] Solaris 2 TCP/IP. http://www.sun.com/sunsoft/solaris/networking/tcpip.html.

[31] Squid. http://squid.nlanr.net/Squid/.

[32] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Labs., May 1995.

[33] thttpd. http://www.acme.com/software/thttpd/.

[34] B. Totty. Personal comm., Mar. 1998.

[35] Zeus. http://www.zeus.co.uk/.