Better operating system features for faster network servers

Gaurav Banga gaurav@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Peter Druschel druschel@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Jeffrey C. Mogul mogul@pa.dec.com
Digital Equipment Corp. Western Research Lab., 250 University Ave., Palo Alto, CA, 94301

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, priority inversion, 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 system interfaces and algorithms and the requirements of modern server applications. Most current operating systems (except for single-user desktop systems) were designed either for efficient timesharing, or for database or file service. In such applications, 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 I/O 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 UNIX.

In most operating systems, 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 [26, 29], 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 [27, 38], 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 [11, 39, 42, 44] use a single-process event-driven architecture, to reduce context-switching overhead. Even these servers have some scalability problems [6, 18, 25].

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, 32, 36, 40].

2 Evolution of Internet 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 Internet servers such as proxy, mail, file, and directory 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 [33]) used a set of pre-forked processes. In this model, a master server process accepts new connections and passes them to the pre-forked servers, using UNIX domain sockets.

The next innovation eliminates the master process. Instead, each pre-forked server calls accept() directly to accept new connection requests. The Apache server [4] has this architecture.

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 [11, 39], Zeus [44], thttpd [42] and several research servers [5, 24, 36] all use an event-driven architecture.

Another alternative is the single-process multi-threaded architecture. In a simple multi-threaded server, 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. These pages are subsequently transferred to the client through the Web server. To make the construction of such auxiliary programs easier, several standard interfaces that govern the communication between Web servers and such programs have been defined. Examples include CGI [9] and FastCGI [16]. The earlier interface, CGI, creates a new process to handle each dynamic document request. The newer FastCGI allows persistent dynamic document server processes. Microsoft and Netscape have also defined new library interfaces [21, 34] to allow the construction of third-party components that can reside in the main server process, when fault isolation is not an issue.

Internet 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 implements the functionality to serve all static documents. Dynamic documents are created by either library code within the main server process, or by auxiliary processes whose code needs to be kept apart from the other components of the server for reasons of fault isolation. In a sense, this is ideal because 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 either event-driven or multi-threaded servers. In this section, we describe these inadequacies in detail, and 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 mechanisms 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 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 (for example, using SIGIO to indicate an I/O completion) 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 multiple 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 established connections, to discover which descriptors 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 tens to hundreds of milliseconds. This prevents any progress, even on unrelated connections that could be handled without additional I/O. Although event-driven servers on fast SMP hardware today can otherwise handle at least 7214 requests/second [41] (i.e. a request every 139 μs), even moderate amounts of disk I/O can degrade performance to disk speed, i.e. 60-120 requests/sec.

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]. Pure 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 user-level thread blocks on an I/O event, all of the threads in the process stall. This is analogous to the blocking I/O problem in event-driven servers.

To rectify these problems, Anderson et. al. [3] proposed "scheduler activations." Scheduler activations are kernel thread-like schedulable entities which 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 in the kernel, the kernel provides another scheduler activation so that other unblocked user-level threads can continue to run. Thus context-switching is fast, and a blocked user-level thread does not cause all other threads of the process to stall.

Operating system thread support has other limitations peculiar to Internet servers. A multi-threaded Internet 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 technical challenge is to minimize the context-switching overhead and the TLB miss rates that result from maintaining a large number of thread stacks. Excessive synchronization overhead can also be a problem. It is because of these overheads that some servers, such as the Inktomi traffic server, use a hybrid control model [43]. Such servers use a moderate number of threads, each of which is an event-driven state machine. This gets around some problems of pure event-driven servers, such as blocking disk 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, such as the AltaVista front-end [8], 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 [10, 15, 19] 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 operating system 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 [14, 30]. This is particularly important for servers because they are, by their nature, particularly network intensive.

The LRP network subsystem architecture [14] was designed to address these problems by more closely following the process-centric model. In this architecture, network processing is correctly integrated into the system's global resource management. Resources spent in processing network traffic are associated with and charged to the application process that caused the traffic. Incoming network traffic is scheduled at the priority of the process that received the traffic, and excess traffic is discarded early. LRP systems exhibit better fairness between applications, and provide stable overload behavior.

However, even such a "faithful" implementation of the process-centric resource model fails to support single-process Internet 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, and 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 rates of connections. This is because most of the work performed in processing network packets is done in the kernel, and cannot be controlled by the application.

For an event-driven server, this implies that the server process cannot control the system resources consumed by the various open connections. These resources include CPU time, network buffers, etc. Nor can the server application control the order in which the kernel delivers network events. Thus the server cannot control the progress rates of its connections.

A multi-threaded server can exert some control over the user-level CPU resources consumed by a particular connection, by adjusting the relative priority of the connection's thread. However, as in the event-driven case, the kernel-mode CPU consumption of a particular connection is uncontrolled. Again, this is because the kernel does not distinguish between the protocol processing activity that it performs on behalf of different threads of a process. Protocol processing for all threads is performed in either the context of a software interrupt (as in vanilla UNIX), a generic kernel thread (as in, for example, Digital UNIX), or by a per-process thread (as in LRP). Moreover, the consumption of other resources, such as buffer space, is also uncontrolled.

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 QoS support.

Under overload conditions, this 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. This instability makes 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, in the limit a good event-driven implementation might perform better than a good thread-based implementation. We base this expectation on reasons given by Ousterhout [35], including the necessity for locking and context-switching in a thread-based system. Also, an event-based program can use a single execution stack; a thread-based program uses multiple stacks, putting more pressure on the data caches and TLB.

However, even Ousterhout admits that threads are a more powerful abstraction, and argues against them primarily because of programming complexity. We also 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. Because existing operating systems do not optimally support large-scale servers using either threads or events, we cannot yet use experiments to decide which model inherently provides better performance.

Certainly threads are necessary to exploit the full power of a multiprocessor. A hybrid model, using a moderate number of threads and an event-based notification mechanism, may be best for Internet servers.

As mentioned in Section 3.2, the issues involved in efficiently supporting threads are relatively well understood [3, 10, 19]. 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*, which encompasses all system resources that the server uses to service a particular client connection. All kernel processing for a particular connection is charged to the appropriate resource container, and scheduled at the priority of the container.

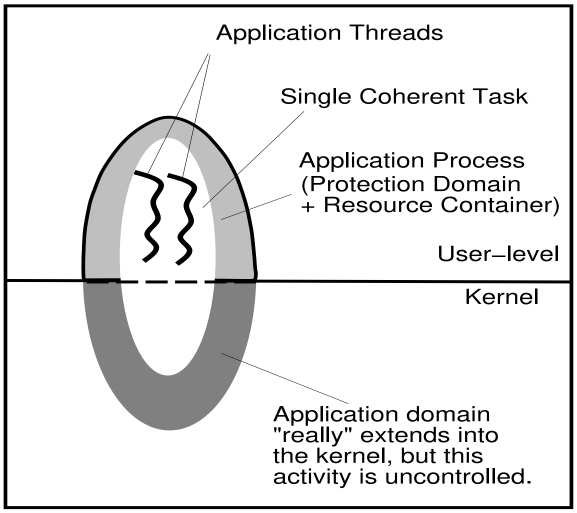

Figure 1: A classical network intensive application.

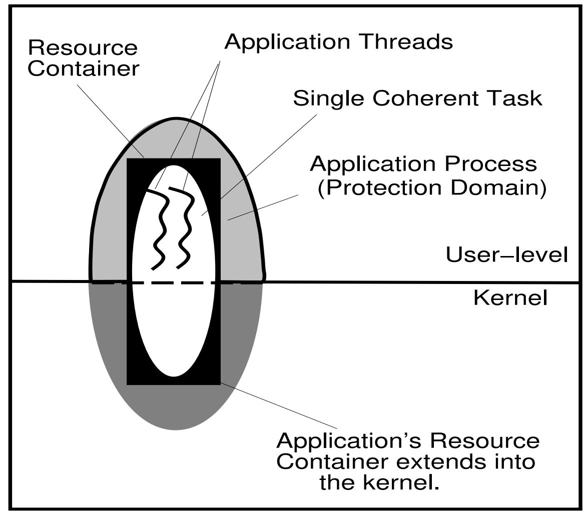

Figure 2: A network intensive application in a LRP system.

Each resource container has an associated priority value, 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 resource container. The application process 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 none at all; it is more nearly true using a mechanism such as LRP [14].) In current systems, a process has a dual function: it serves as a protection domain, and as a resource container. The protection domain aspect of a process provides a mechanism for isolation between applications. The resource container aspect of a process, on the other hand, provides the resource management subsystem of the operating system with resource principals between which the system resources are to be shared. Unfortunately, this equivalence between protection domains and resource containers 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 current process abstraction suffices.

Figures 1 and 2 depict this situation. Figure 1 shows a classical application, which uses a single process to perform a single network-intensive coherent task. As described in Section 3.3, the kernel resource consumption of such applications in classical systems is largely uncontrolled. LRP extends a process's resource-container into the kernel leading to the situation shown in Figure 2. Note that even in LRP, the resource container abstraction is still firmly linked with the process abstraction.

Sometimes a single application, performing a single coherent task, is split up into multiple protection domains. Reasons for this include, for example, the provision of fault isolation between the different components of an application, or the use of components supplied by several vendors. For these applications, the desired unit of protection (the process) is different from the desired unit of resource management (all the processes of the application). A mostly user-mode multi-process application trying to perform a single coherent task is shown in Figure 3.

In yet another scenario, an application consists of a single process, and yet tries to accomplish multiple independent coherent tasks. Such applications use a single protection domain in order to have lower context-switching overhead. For these applications, the correct unit of resource management is smaller than a process: it is the set of all resources being used by the application to accomplish a single task. Single-process Internet 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 really a combination of the last two scenarios. Usually, a single server process manages a large number of connections. Sometimes, however, the situation resembles a multi-process scenario. This happens, for example, when the main server process forks off a CGI script to handle a dynamic document request, or when it retrieves a dynamic document from a persistent CGI

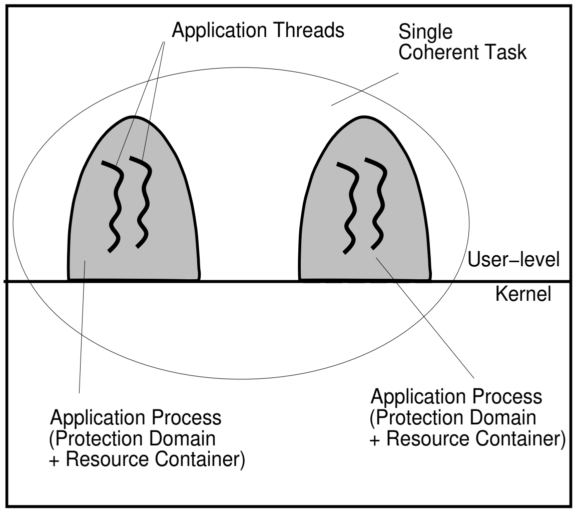

Figure 3: A classical multi-process application.

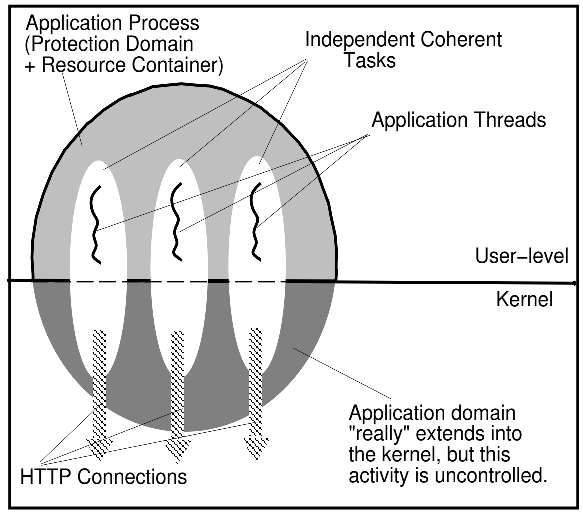

Figure 4: A single process multi-threaded server.

server [36]. In both cases, the desired unit of resource 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 equivalence between "process" and "resource container" 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 adequately accomplished using user-level mechanisms, because kernel resource utilization forms an insignificant part of the total resource usage of the process. As noted in Section 3.3, 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 resource container, and to create various types of association between threads and resource containers.

Resource containers are named within a process by file descriptors. While we could instead have defined a new namespace for resource containers, the use of of file descriptors allows the use of several existing system calls to manipulate resource containers. For example, the sendmsg() system call can be used to transfer file descriptors, and hence resource containers, between protection domains.

The kernel execution model in the new system is a generalization of the LRP approach [14]. Like in LRP, a variety of methods can be used to execute kernel code. For instance, a dedicated per-process kernel thread can be used to perform all kernel processing for each process. We are currently in the process of building and refining these mechanisms and interfaces.

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

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

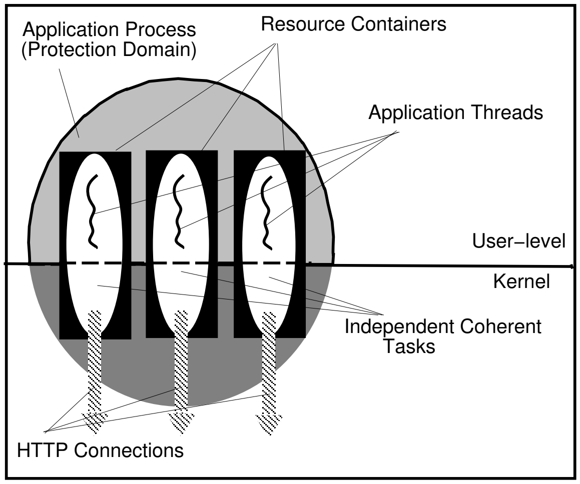

Figure 5: Resource containers in a multi-threaded server.

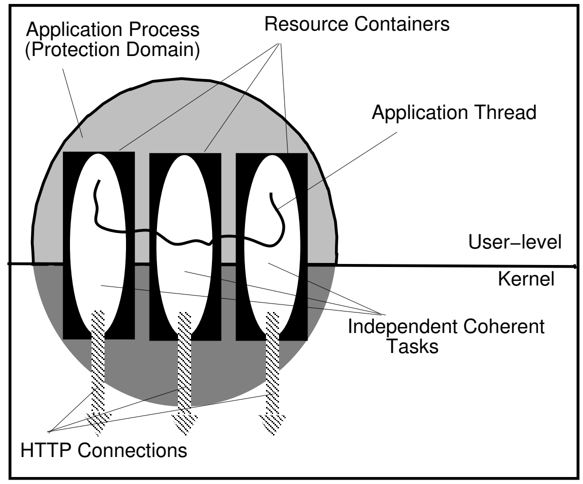

Figure 6: Resource containers in an event-driven server.

If any connection consumes a lot of resources, this manifests itself as higher resource usage counter values for the corresponding container. We provide an application programming interface (API) to allow a process to obtain resource usage for a container. 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 multiple user-level threads may employ resource containers in a similar manner.

In both kind of servers, when a request for a CGI document comes along, the connection's container is passed to the process which creates the dynamic document. If traditional CGI is being used, in which a new process is forked for each CGI request, this container is simply inherited by the child process. If persistent CGI is being used, the connection's container is passed to the CGI server 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 not only the local port number and Internet address, but also 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 [37]. 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 then assigns all requests from a particular client, or set of clients, to just one of these sockets. By associating a different resource 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 (ISP) can use this to control the priorities at which different classes of clients are handled.

Placing the 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 being overloaded by a denial-of-service attack from a malicious client.

Resource containers have some properties that makes them similar to a number of resource management mechanisms that have been developed in the context of some recent experimental operating systems [23, 31]. The key factors that distinguish the resource container abstraction from these mechanisms are its generality and direct applicability to current, general purpose operating systems. A comparison of resource containers and other mechanisms is deferred until Section 5.

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 one or more descriptors. The revoke_interest() system call indicates that it is no longer interested in events on a set of descriptors. For example, when a server acccepts a new connection, from which it will read a request message, 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. This indicates interest in the disk I/O completion event.

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.

There are other viable alternatives to the proposed dequeue_next_events() system call. For example, event notifications can be structured as upcalls [12], with the server process specifying handlers for various types of events directly to the operating system kernel. Similarly, the kernel could indicate events to the server process by using a *pending-events* queue in a memory region shared with the application process. While these approaches might perform better than one that uses a system call, we feel that the system call approach is closer to the APIs provided by current operating systems. For this reason, this approach might be easier to use, and reason about, than one based on upcalls or shared memory. Moreover, unlike the other two approaches it avoids the need for explicit synchronization on the programmer's part. This is important, since one of the primary advantage of an event-driven approach vis-a-vis a multi-threaded one is that the former does not require the complexity of explicit synchronization.

5 Related Work

As mentioned in Section 4.1, 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 [31] has explicit support for a *path* abstraction, which allows an application to control resource consumption for a given communication path at all levels of the system. Paths are similar to resource containers; however, resource containers are more general as they can encompass several otherwise "unconnectable" paths.

Moreover, the binding between a resource container and the kernel resources that can be associated with it is more dynamic and flexible than the association between a path and Scout kernel entities. This is because, in Scout, paths are specified at kernel build time. They can be instantiated, extended, optimized and associated with execution entities (threads) at run time during the path creation phase. However, the association between a path and the system resources associated with it cannot be arbitrarily changed *after* the path creation phase. For instance, we cannot change the binding between a path and a kernel resource, such as a socket, after the path creation phase. Also, unlike resource containers which can encompass arbitrary sets of resources specified at runtime, the composition of a path is limited to the router graph specified at kernel build time.

Scout is a special-purpose operating system built from scratch to efficiently support network appliances. The path abstraction is not available in general-purpose operating systems, and there has been no attempt to integrate paths into current operating system interfaces.

Kaashoek et. al. [23, 24] advocate a customized operating system tailored specifically for servers. In a server operating system based on the Exokernel, the application controls essentially all of the protocol stack, including the device drivers, through a combination of library code and a novel kernel architecture. The Exokernel provides a similar interface to the storage system. This allows the application to directly control the resource consumption for all associated network communication and file I/O. A prototype Web server system that the Exokernel project has built uses several aggressive optimizations to achieve an order of magnitude performance improvement over a server running on a conventional operating system.

The Exokernel approach to operating system development is a radical departure from how current operating systems are implemented. For this reason, this approach represents a point in the design space of server-oriented operating systems that is unlikely to be of immediate utility. Moreover, implementing the Web server system on a Exokernel brings into the domain of the Web server developer several software engineering issues related to the problem of developing and maintaining a complicated library operating system.

The new operating system features that we propose allow many of the benefits of the Exokernel's application-controlled resource management to be achieved in the context of general purpose operating systems.

At a superficial level, the functionality provided by resource containers is similar to that provided by a number of operating system abstractions developed in the context of multimedia and real-time operating systems. These include the processor capacity reserves of Mercer et. al. [28], the *activities* [22] of Rialto, the *migrating threads* of Mach [17] and AlphaOS [13], and the *shuttles* of Spring [20]. The chief differences are related to the more general nature of resource containers. Processor capacity reserves and activities are real-time abstractions and are thus more complex to implement than resource containers. Also, migrating threads, processor capacity reserves and shuttles are micro-kernel specific solutions which do not address the problem of controlling the resource consumption of kernel I/O processing.

Almeida et al. attacked the problem of providing QoS support in a Web server running on a widely available general-purpose operating system [1]. They mapped QoS requirements onto scheduling priorities, experimenting both with a user-level implementation, and with a slightly modified Linux kernel scheduler. They used the Apache server [4], and so followed the process-per-connection model, although their approach could probably be extended to a thread-based server. They found that this approach allowed them to provide differentiated service to HTTP requests in different QoS classes, albeit with some limitations on effectiveness. However, they did not evaluate how accurately their system allocated kernel-mode time, and their implementations did not even attempt to set priorities for processing of received packets, or to differentiate between existing connections and new connection requests.

6 Summary

We discussed the need for efficient operating system support for Internet servers, and identified two dimensions of this support. First, the operating system needs to allow the process to manage the kernel's consumption of resources for individual connections. We therefore proposed the *resource container*, a kernel-supported abstraction binding together 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 the two common server execution models, thread-based and event-driven. We proposed a new API for event handling, which should scale to large numbers of connections per process.

We are currently working on a prototype implementation of our events API and resource container abstraction. We are also continuing to improve our understanding of the limitations of current operating system support for large-scale multi-threaded applications.

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, 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] E. W. Anderson and J. Pasquale. The Performance of the Container Shipping I/O System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, page 229, Copper Mountain, CO, Dec. 1995. ACM.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109. Association for Computing Machinery SIGOPS, October 1991.

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

[5] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 USENIX Technical Conference*, Jan. 1997.

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

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

[8] M. Burrows. Personal communication, Mar. 1998.

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

[10] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[11] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, Jan. 1996.

[12] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Dec. 1985.

[13] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 127–146, Seattle WA (USA), Apr. 1992. Usenix.

[14] 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.

[15] D. R. Engler, D. K. Lowenthal, and G. R. Andrews. Shared Filaments: Efficient Fine-Grain Parallelism on Shared-Memory Multiprocessors. Technical Report TR 93-13a, University of Arizona, CS Dept., Tucson, AZ, 1993.

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

[17] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the 1994 Winter Usenix Conference*, Jan. 1994.

[18] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[19] D. Grunwald and R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. In *Proceedings of the* 2nd Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, Oct. 1996.

[20] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. of the 1993 Summer Usenix Conference*, pages 147–159, Cincinatti, OH, June 1993.

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

[22] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[23] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sizteenth Symposium on Operating System Principles*, Saint-Malo, France, Oct. 1997.

[24] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *1996 SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[25] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[26] 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.

[27] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming tcp packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–280, Aug. 1993.

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

[29] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[30] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.

[31] 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.

[32] E. Nahum, T. Barzilai, and D. Kandlur. Evaluating High Performance Socket APIs on AIX. *submitted for publication*, Feb. 1998.

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

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

[35] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference. http://www.scriptics.com/people/john.ousterhout/threads.ps.

[36] 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.

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

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

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

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

[41] The Standard Performance Evaluation Corporation. SPECweb96 Results. http://www.specbench.org/osg/web96/results/, Apr. 1998.

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

[43] B. Totty. Personal communication, Mar. 1998.

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