Server Operating Systems

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach
M.I.T. Laboratory for Computer Science
{kaashoek,engler,ganger,kerr}@lcs.mit.edu

Abstract

We introduce server operating systems, which are sets of abstractions and runtime support for specialized, high-performance server applications. We have designed and are implementing a prototype server OS with support for aggressive specialization, direct device-to-device access, an event-driven organization, and dynamic compiler-assisted ILP. Using this server OS, we have constructed an HTTP server that outperforms servers running on a conventional OS by more than an order of magnitude and that can safely timeshare the hardware platform with other applications.

1 Introduction

Servers, the foundations of the client/server model of computing, are rapidly becoming more critical. If we are to realize the promise of global information access, widely distributed computing and even high-performance local area computing, system implementors must construct a variety of server applications (whether centralized or not) that can support large numbers of active clients. Ideally, the development and operating environments should be conducive to easy and modular construction of servers (e.g., HTTP, FTP, NFS, etc.) that deliver the performance of the underlying hardware (in particular, the network and/or disks) without requiring that an entire machine be dedicated to each server. Unfortunately, the status quo falls short of this ideal.

Currently there are two main approaches to constructing servers. The first is to place a server on top of a general-purpose operating system (OS). This approach can simplify the construction of server applications, but seriously compromises performance by forcing them to use overly general OS abstractions. These abstractions frequently provide an order of magnitude less performance (for primitive operations) than is available from the hardware [2, 8, 10, 13, 22, 23]. Furthermore, these abstractions are usually high-level, directly preventing servers from exploiting domain-specific knowledge. With this approach, achieving high performance generally requires very powerful hardware (e.g., Alta Vista [7], which uses 12 state-of-the-art Alpha CPUs and over 7 GB of physical memory). The second approach is to create an operating system specifically designed for a single server configuration [11, 12, 20]. With this approach, a different operating system is generally constructed from the ground up for each different server, greatly increasing the implementation effort [20]. Furthermore, because this approach does not multiplex resources among multiple servers, it requires that each server have an entire machine dedicated to it, even though server traffic tends to be bursty and could therefore be multiplexed. As a result, this second method is also costly and exhibits poor resource utilization, which ultimately compromises performance. We believe that we can have the best of both worlds: the specialization and performance of a dedicated system with the modular construction and inter-application protection of a general-purpose OS.

We propose a third approach: *server operating systems*. A server operating system is a set of abstractions and runtime support for specialized, high performance server applications.[1] A good server operating system should provide (1) tools and parameterizable default implementations of server abstractions (e.g., network protocol implementation, storage management, etc.) to support modular construction of server applications, (2) full freedom to replace or override these default implementations and specialize server abstractions based on application-specific characteristics, and (3) protection boundaries, such that multiple applications (including both highly specialized servers and "normal" applications) can timeshare a high-performance system effectively.

[1] We talk about server OSs in the plural, because different server OS support may be appropriate for different types of server applications.

We are in the process of constructing a server operating system that includes:

- Default implementations of various abstractions useful for constructing server applications, implemented in such a way that they can be parameterized and combined in application-specific ways (e.g., application-specific disk layouts, header and checksum precomputation, etc...).
- Support for direct, protected access to hardware resources, allowing a server application to completely replace any or all of the default abstractions with its own when none of the existing choices matches its needs.
- Support for direct access from the disk system to the network module (and vice versa), avoiding scheduling delays, traversal of file system layers and network layers, and redundant data copies.
- Support for event-driven organization of server applications, which avoids the thread management and concurrency control problems inherent in a thread per request organization.
- Compiler-supported, dynamic integrated layer processing (ILP) [6] to improve the throughput of application-specific networking software.

The construction of server OSs is greatly simplified by using an extensible operating system. We are constructing our prototype server OS as a set of libraries on top of the Aegis exokernel [8], which provides applications with direct, protected access to hardware resources. The exokernel OS architecture, by design, allows arbitrarily-specialized applications to safely timeshare a system. In designing and implementing server OSs, we view this support as a given and focus on identifying abstractions and building libraries that simplify the construction of highly specialized server applications that deliver the full hardware performance. Although we use the exokernel as a platform for building server OSs, it is likely that other extensible operating systems (e.g., Spin [2], Cache Kernel [5], Vino [23]) could also provide the necessary base support. Some of the extensible abstractions that we are developing could even be added to a conventional OS (e.g., UNIX or Windows NT).

The contributions of this paper are threefold. First, we describe and argue for server operating systems as a better way of constructing high performance server applications. Second, we identify design techniques for server OSs and discuss how to apply these in a modular fashion. Third, we describe a prototype server OS and its use in constructing a high-performance HTTP server. Measurements of this server show that it can support an order of magnitude more client activity than conventional implementations without sacrificing inter-application protection.

The remainder of this paper is organized as follows. Section 2 describes the design of the components of a prototype server operating system that we are constructing. Section 3 describes our initial implementations of these components. Section 4 describes a fast HTTP server (called Cheetah) constructed with the prototype server operating system. Section 5 presents measurements of Cheetah to show that a server application constructed in this way indeed delivers high performance. Section 6 discusses related work and Section 7 summarizes the paper.

2 A Server Operating System Design

In this section, we describe in more detail the components of a prototype server operating system that we have designed and are in the process of building.

2.1 Specialization

It has been clearly demonstrated that specializing abstractions and their implementations in application-specific ways can substantially increase performance (e.g., see [2, 8, 13, 22]). Because performance is critical for server applications, our prototype server OS supports specialization directly. At the same time, because of the extensible OS used as a base, multiple applications can safely co-exist on a system, even when some of them use differently-specialized resource management policies.

To reduce the software engineering effort involved, our prototype server operating system provides a variety of parameterizable implementations of abstractions appropriate for high-performance server applications, including a file system that allows specialized on-disk layouts, a disk cache that allows specialized replacement/prefetching policies and a network protocol (TCP/IP) implementation that allows specialized message control. Of course, to support arbitrary specialization, our prototype server OS also allows server applications to use their own resource management abstractions instead of any of the default implementations provided.

2.2 Direct device-to-device access

The main task of many servers is moving data from the storage subsystem to the network (and vice versa). Therefore, it is critical that this data path be as streamlined as possible. In theory, it should be possible to minimize software overhead such that the bandwidth of this path is limited only by the slower of the two devices. To realize this ideal, a server application must eliminate scheduling and notification delays, file system layer and network layer traversal, and redundant data copies from the critical path.

Our prototype server OS therefore enables server applications to easily integrate the control and data flows of hardware devices. For example, one useful abstraction provided is a combined disk cache and network buffer, which eliminates both duplication and copying between the two. Also, device-to-device data movement is made more efficient by allowing application-specified network interrupt handlers to initiate activity in the disk system (and vice versa). For example, an application-specified handler routine for disk I/O might initiate a network transmission immediately after a disk read completes.

2.3 Event-driven organization

By nature, server applications are reactive, responding to client requests rather than having their own sequence of actions to take. Each client request generally involves several sequenced I/O events (e.g., network packet arrival, disk request completion, etc.) with small amounts of computation (relative to I/O delays) interspersed. For acceptable performance, it is generally necessary to service multiple client requests concurrently to exploit parallelism in the I/O system. Many current server implementations use a separate thread (or process) per request and conventional blocking I/O to do this. Unfortunately, this approach can significantly increase complexity and decrease performance due to thread creation/deletion, thread switching, data sharing and locking.

Another approach is to use non-blocking I/O and a general event loop abstraction (provided by the server operating system), such that a server application can simply consist of a set of event handlers that react to external stimuli by initiating additional I/O actions (e.g., disk requests or network transmissions). With such an event-driven organization, a server application can exploit the same level of concurrency (on a uniprocessor) without the problems described above [18]. Our prototype server OS supports non-blocking versions of file system and network protocol abstractions to support event-driven programming.

To make this organization even more effective, our prototype server operating system allows aggressive server applications to construct and install code modules (called *application-specific safe handlers (ASHs)* [24]) to be executed immediately upon the occurrence of a relevant hardware interrupt. So, for example, when a network packet or disk interrupt destined for the server application is taken, an application-constructed handler can be run (in addition to the kernel code required for dealing with the interrupt). Such a handler may update server application state or initiate network transmissions or disk requests, as appropriate. If necessary (e.g., because longer processing or a held lock is needed), notification of the interrupt can be propagated to the server application proper. Although ASHs do increase complexity by introducing additional threads of control (thereby requiring some locking), they allow common cases to be handled with minimal delay/overhead, offering substantial performance improvements.

2.4 Dynamic, compiler-assisted ILP

One important, well-known technique for improving network software performance is integrated layer processing (ILP), wherein multiple protocol actions (e.g., checksum and byte-swap) are merged into a single-pass, minimal-overhead function [6]. In particular, by combining the protocol actions into a single pass over message data, ILP can reduce the impact of memory system performance on networking.

Unfortunately, achieving near-optimal ILP manually is a non-trivial task that needs to be re-addressed each time the sequence of protocol actions changes. Therefore, because high performance network software is so important to many server applications, our prototype server OS provides compiler support for ILP. In particular, an application writer need only specify each individual data manipulation step and their sequence. The ILP compiler can then automatically perform the integration (a.k.a., composition), which, in most cases, it can do at least as well as an application writer. Protocol specification is further simplified by providing libraries of common data manipulation steps needed for protocol actions.

The compiler-assisted ILP mechanism also supports dynamic (i.e., run-time) integration. One example of why dynamic ILP is useful is ethernet cards that have different rules for how data are moved to/from their buffers. With static ILP, all combinations of different rules and the remainder of the integrated protocol must be constructed and linked. With dynamic ILP, the one relevant combination can be constructed at run-time.

3 Implementation of a Prototype Server Operating System

Our goal in constructing server operating systems is to enable modular construction of server applications that deliver full hardware performance, without artificial limitations caused by unnecessary software overheads. This section describes our ongoing implementation of a prototype server OS, which includes efficient and parameterizable implementations of TCP/IP [21] and a disk-based file system, in terms of the server OS support outlined in Section 2.

We are building server OSs as libraries on top of an exokernel [8]. The exokernel OS architecture is designed to provide application-level software with direct, protected access to hardware resources by limiting kernel functionality to multiplexing hardware resources among applications. This kernel support is sufficient to allow us to construct server OSs such as the one described in Section 2. In addition, because the exokernel OS architecture allows multiple differently-specialized applications to safely co-exist and timeshare hardware resources, we can focus our server OS implementation on providing useful abstractions and reducing software engineering complexity. As a beneficial side-effect of the exokernel architecture, the fact that the server OS is in application space generally makes it easier to build, test and debug.

Specialization To support modular specialization, we have made our TCP/IP and file system libraries highly parameterizable and easy to integrate with other components of a server application. For example, the TCP/IP library does not manage protocol control block (PCB) allocation, allowing server applications to incorporate PCBs into their own central data structures. It also allows applications to specify that a particular transfer is the last for the connection (so that the FIN flag can be set in the last data packet, instead of sending a separate packet) and to provide precomputed checksums for the data being transmitted.

The file system library implements a file system similar to the Fast File System [14]. However, in addition to support for non-blocking operations, as described above, this library is highly configurable. In particular, the cache replacement, miss handling, write-back and flush routines are all specified by the application during the initialization phase. Although default implementations exist, it is trivial to replace them. Also, the disk allocation code can be replaced easily, allowing for application-specific data layouts. Finally, extra inode fields are provided to allow server applications to add their own information (e.g., prefetching hints and extra type information).

Direct device-to-device access The TCP/IP library and the file system library both support direct device-to-device data movement by allowing applications to use scatter/gather I/O and specify source (destination) locations for outgoing (incoming) data. In addition, the TCP/IP library does not keep retransmission buffers, but instead invokes a call-back if the data must be retransmitted. Together with file system support for pinning and write-locking disk cache blocks, this allows applications to construct a combined, copy-free disk cache/retransmission pool.

Applications can use the support provided by the TCP/IP and file system libraries to easily construct a data path between network and disk that involves zero memory-to-memory copies, unless such copying is required by the device controller implementations. Unfortunately, the Ethernet cards in our current experimental platform use programmed I/O to move data to and from dedicated buffer memory. We are also starting to experiment with disk to network data movement over an ATM network (Digital's AN2) with a card that uses DMA to move data to and from arbitrary physical main memory locations. Using the combined disk cache/retransmission pool, precomputed checksums (described below) and scatter/gather I/O to add the headers, we expect to see that the CPU never has to touch the data, flush the data caches (after DMA input or before DMA output), or even map the corresponding physical memory when moving data over TCP/IP connections from the disk to the network.

Event-driven organization The TCP/IP and file system libraries support an event-driven organization of server applications by providing non-blocking interfaces and assisting with correlation of I/O events and previously saved state. The non-blocking interfaces progress as far as possible and then return control to the application with an indication of how much progress was made. To assist with identifying context for an I/O event, the non-blocking interfaces save application-specified values, demultiplex events internally and return the corresponding value to the application.

Dynamic, compiler-assisted ILP We use *pipes* [1] to provide dynamic ILP. A pipe is a computation written to act on streaming data, taking several bytes of data as input and producing several bytes of output while performing only a tiny computation (such as a byte swap, or an accumulation for a checksum). Our pipe compiler can integrate several pipes into a tightly integrated message transfer engine which is encoded in a specialized data copying loop. The pipes are written in a low-level language similar to that of a RISC machine, augmented with useful primitives (such as byteswap). The dynamic ILP interface and implementation are described in [9].

4 Cheetah: A Fast HTTP Server

This section describes Cheetah, a fast HTTP server constructed using our prototype server operating system. The development of Cheetah proceeded in several stages, which we expect to be the common approach when building highly-specialized server applications. We began with a simple, non-specialized implementation (actually developed under SunOS) linked against the default POSIX-interface libraries provided for the exokernel system. We then replaced the major components (TCP/IP and file system), one at a time, with those from the server operating system. Finally, we added specializations, again one at a time, progressively improving Cheetah's performance. Although Cheetah does not yet use ILP or ASHs, our measurements show that it achieves an order of magnitude improvement in performance over other popular HTTP servers (without sacrificing the freedom to timeshare the hardware platform).

4.1 Overview

Most of Cheetah's implementation consists of configuring and tying together the TCP/IP and file system libraries provided by the server operating system. The combined disk cache and retransmission buffer implementation described in Section 3 is used to eliminate this source of data copying and wasted physical memory. Each ongoing HTTP request is represented by a single data structure that includes the request status, the TCP protocol control block (PCB), a TCP/IP send header, and various request state. Cheetah executes a single thread of control that, after a configuration/initialization phase, repeats an endless loop of polling for network or disk events (by calling the appropriate library functions) and servicing them (with HTTP specific functions, such as request parsing, and non-blocking library functions). When the processing of an HTTP request reaches the point that it must wait for some external stimulus, its critical state is saved in the request structure and control returns to the main event loop.

4.2 TCP/IP specialization

Although our TCP/IP library implementation is tuned for a high rate of short-lived connections (e.g., by minimizing the impact of connections in the TIME_WAIT state), as characterizes an HTTP server, Cheetah attempts to further reduce connection establishment costs. For example, it keeps a pool of pre-allocated, pre-initialized HTTP request structures (including PCBs) to avoid the need for most allocation during setup. Also, Cheetah uses the memory that contains the HTTP request message when it arrives, avoiding allocation and data copying for most cases.

Because web pages tend to be small, it is important to minimize the number of separate network packets to reply to an HTTP request. In particular, a conventional implementation will use separate packets to send the HTTP header, the request data and the FIN flag. Cheetah uses the scatter/gather TCP/IP library interface with FIN specification to combine these three pieces of information into the smallest number of packets allowed (given the maximum segment size negotiated by the client and the server).

Cheetah also uses precomputed TCP/IP checksums when sending web page contents. These checksums are stored on disk with the corresponding file and computed by the file system only when the file is modified. Because popular web pages are generally accessed many times without being changed, this approach both moves the checksum computation out of the critical path and amortizes it over multiple accesses.

4.3 File system specialization

As with precomputed checksums, Cheetah precomputes the HTTP response header for each file and stores it with the file. With the exception of the date, which is computed separately (in the background, when time advances) and added by Cheetah at the last moment, none of the header for a `GET` or `HEAD` HTTP request changes unless the file is modified. For `GET` requests modified by the `IF-MODIFIED-SINCE` MIME header, only the return code value and message changes if the condition evaluates to false. Precomputing the response header reduces the work that must be done in the critical path, eliminating costly time conversions, string generations and string comparisons (for determining MIME types). It also enables precomputation of the checksum over the HTTP header.

The hypertext links found in HTML pages can be exploited at disk allocation time, disk fetch time and cache replacement time to improve performance. Because most browsers fetch inlined images immediately after fetching a page, placing such files adjacent to the HTML file and reading them as a unit (effectively prefetching the images) can improve performance. Likewise, this relationship can be exploited during cache replacement.

5 Experimental Evidence

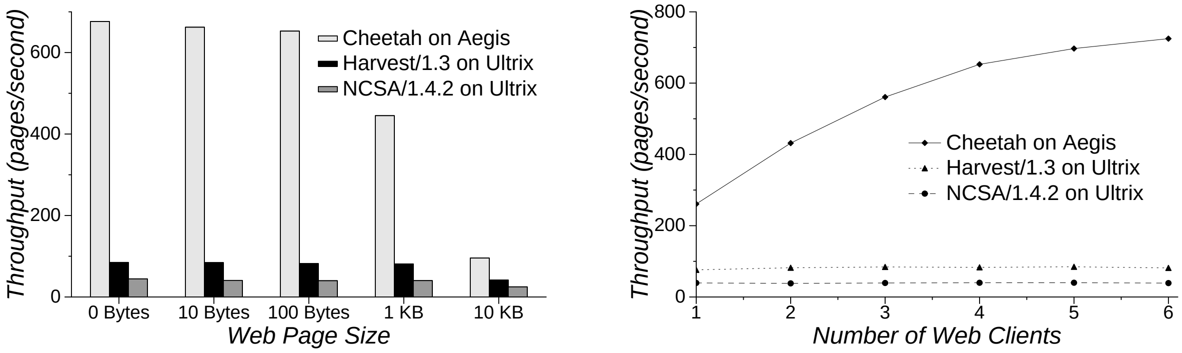
This section compares Cheetah to both an NCSA server running on top of Ultrix and the Harvest httpd-accelerator acting as a front end to the NCSA server. The results show that Cheetah's aggressive specialization improves its performance by more than an order of magnitude for high request rates and small document sizes. We expect the performance of Cheetah to increase further when we exploit the ILP and ASH techniques described earlier, as both have been shown to offer reductions in software overhead [6, 8, 24].

For all of the experiments, the HTTP servers were running on dedicated DECstation 5000/125s, which are 25 MHz MIPS R3000 machines. Each client application synchronously requests a single document (i.e., URL) 1000 times. For the experiments, one instance of this application was run on one or more other machines (Sun workstations running SunOS and Pentium-based systems running Open BSD) connected to the server systems via 10 Mbit/second Ethernet. The experiments were run when no other users were present. As discussed earlier, Cheetah runs on the Aegis exokernel. For comparison purposes, we measured both NCSA/1.4.2 [16] and Harvest/1.3 [4] (configured as an httpd-accelerator) running on Ultrix v4.3.

Figure 1a shows the throughput (measured in documents served per second) supported by the three HTTP server configurations. For small document sizes (0 bytes, 10 bytes and 100 bytes), Cheetah services 8 times as many requests as the Harvest cache, which in turn serves 2 times as many as the NCSA server alone. For 1 KB and 10 KB documents, network limitations reduce the difference between Cheetah and Harvest to factors of 5.5 and 2.3, respectively. With 10 KB documents, Cheetah is delivering 970 KB/s of useful document data (i.e., ignoring HTTP headers and TCP control messages) over the 10Mbit/s Ethernet.

Figure 1b shows the throughput (for 100 byte documents) supported by the three HTTP servers as a function of the number of clients. Performance for both the NCSA server and the Harvest cache is roughly independent of the number of clients. Given the performance of Cheetah, this indicates that server software overhead is the bottleneck for these configurations. For Cheetah, performance increases with the number of clients, indicating that the server is not the bottleneck. With one client, Cheetah services 261 requests per second (for an average end-to-end response time of 3.8 ms), which is 3.4 times more than are serviced by the Harvest cache. With six clients, Cheetah throughput exceeds that of Harvest by a factor of 9.

All of the Ultrix server numbers up to this point represent the throughput observed for the first 100 requests, as opposed to sustained throughput. Despite the aggressively performance-conscious implementation of the Harvest



(a) Throughput vs. Document Size

(b) Throughput vs. Number of Clients

Figure 1: HTTP server throughtput. Graph (a) is for 4 clients and graph (b) is for 100 byte documents.

cache, which serves it well during light workloads, its performance falls rapidly as the workload increases because of the TCP/IP implementation thrust upon it by Ultrix. (We believe that this reduction is caused by connections in the TIME_WAIT state [15].) After 1500 requests, Harvest throughput drops by 48% and exceeds that of the NCSA server (whose throughput also decreases by 21%) by only 38%. In contrast, Cheetah's performance remains stable in the face of heavy connection rates. Using these values instead of those compared earlier, Cheetah throughput for small documents exceeds Harvest and NCSA throughputs by factors of 17 and 24, respectively.

6 Related Work

Most servers are implemented on general-purpose operating systems (e.g., UNIX, Windows NT). Examples of such are too numerous to list. In practice, the approach taken to boosting performance of such systems is to use faster hardware. Somewhat less common are examples of the other extreme: server software built on or linked into a rudimentary kernel. Network Appliance's approach to building high-performance servers, as illustrated by their very successful NFS systems (e.g., [11]), is to dedicate each system to a single server application running on top of a rudimentary kernel. This approach precludes effective time-sharing of the hardware resources among a server and other servers or maintenance applications. Servers are constructed on Novell's NetWare operating system by linking modules with the kernel [12]. This approach eliminates protection boundaries, making sharing the machine among multiple servers/applications difficult.

Many aspects of the prototype server OS described in this paper have been proposed and/or implemented previously. For example, integrated layer processing was introduced in [6] and a static, compiler-assisted implementation is described in [1]. The event-driven programming style is an old idea, for which John Ousterhout made a strong case in [18]. Both the Harvest cache and Open Market's high-performance WebServer [17] are implemented in this style (to the extent that the OSs on which they operate allow). Support for direct device-to-device data movement is a main focus of both the container shipping mechanism [19] and the Scout operating system [10]. Finally, specialization and support for it in the form of extensible operating systems is a central focus of a number of OS projects, including Synthesis [13], the Cache Kernel [5], Vino [23], Scout [10], SPIN [2], the exokernel [8], and Synthetix [22]. Our contribution is to extend and combine these independent techniques into server OSs that enable a modular approach to building specialized, high-performance server applications.

7 Summary and Conclusions

We have argued for server operating systems, which support modular construction of specialized, high-performance server applications. Our first server OS design includes four main design techniques: (server-specific specialization, direct device-to-device access, event-driven organization, and dynamic compiler-assisted ILP) to ease the construction of server operating systems. Using a prototype server OS, we constructed an HTTP server that provides more than an order of magnitude increase in performance when compared to servers running on a conventional OS while still allowing for safe timesharing of the hardware. Other servers, including NFS [3] and FTP, are under construction.

Server operating systems offer a way of harnessing the substantial performance improvements offered by extensible operating systems for server applications without the software engineering crisis (caused by separate, slightly different, re-implementations for every application) predicted by some. In fact, we believe that server operating systems can simplify the construction of high performance servers in addition to delivering these performance improvements.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[2] B. N. Bershad, S. Savage, et al. Extensibility, safety and performance in the SPIN operating system. In *15th ACM SOSP*, pages 267–284, December 1995.

[3] H. Briceño. Design techniques for building fast servers. In *MIT Student Workshop*, 1996. To appear.

[4] A. Chankhunthod, P. B. Danzig, et al. A hierarchical internet object cache. In *Usenix Technical Conference*, pages 153–163, January 1996.

[5] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *OSDI*, pages 179–193, Nov. 1994.

[6] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM 1990*, September 1990.

[7] Digital Equipment Corporation. Alta Vista. http://www.altavista.digital.com, 1996.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *15th ACM SOSP*, pages 251–266, December 1995.

[9] D.R. Engler, D.A. Wallach, and M.F. Kaashoek. Design and implementation of a modular, flexible, and fast system for dynamic protocol composition. Technical Memorandum TM-552, MIT Laboratory for Computer Science, May 1996.

[10] J.H. Hartman, A.B. Montz, et al. Scout: A communication-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.

[11] D. Hitz. An NFS file server appliance. Technical Report 3001, Network Applicance Corporation, March 1995.

[12] D. Major, G. Minshall, and K. Powell. An overview of the NetWare operating system. In *Winter USENIX*, pages 355–372, January 1994.

[13] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *12th ACM SOSP*, pages 191–201, 1989.

[14] M. K. McKusick, W. N. Joy, et al. A fast file system for unix. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984.

[15] J. C. Mogul. The case for persistent-connection http. In *ACM SIGCOMM 1995*, pages 299–313, August 1995.

[16] NCSA, University of Illinois, Urbana-Champaign. NCSA HTTPd. http://hoohoo.ncsa.uiuc.edu/index.html.

[17] Open Market, Inc. WebServer technical overview. http://www.openmarket.com/library/WhitePapers/Server/, March 1996.

[18] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at 1996 Usenix Technical Conference (slides available at http://www.sunlabs.com/∼ouster/), January 1996.

[19] J. Pasquale, E. Anderson, and P. K. Muller. Container shipping: Operating system support for i/o-intensive applications. *IEEE Computer*, pages 85–93, March 1994.

[20] R. Pike, D. Presotto, et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[21] J. Postel. Transmission control protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[22] C. Pu, T. Autry, et al. Optimistic incremental specialization: streamlining a commercial operating system. In *15th ACM SOSP*, pages 314–324, Copper Mountain, CO, December 1995.

[23] C. Small and M. Seltzer. Vino: an integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard, 1994.

[24] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *ACM SIGCOMM 1996*, August 1996. To appear.