Fast and Flexible Application-level Networking on Exokernel Systems

Gregory R. Ganger[1], Dawson R. Engler[2], M. Frans Kaashoek[3]
Héctor M. Briceño[3], Russell Hunt[4], Thomas Pinckney[4]

March 2000
CMU-CS-00-117

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1] Carnegie Mellon University
[2] Stanford University
[3] Massachusetts Institute of Technology
[4] ExoTec, Inc.

Abstract

Application-level networking is a promising software organization for improving performance and functionality for important network services. The xok/ExOS exokernel system includes application-level support for standard network services, while at the same time allowing application writers to specialize networking services. This paper describes how xok/ExOS's kernel mechanisms and library operating system organization achieves this flexibility, and shares our experiences and lessons learned (both positive and negative). It also describes how we have used this flexibility to build and specialize three network data services: the Cheetah HTTP server, the webswamp web benchmarking tool, and an application-level TCP forwarder. Overall measurements show large performance improvements relative to similar services built on conventional interfaces, in each case reaching the maximum possible end-to-end performance for the experimental platform. For example, Cheetah provides factor of 2–4 increases in throughput compared to highly-tuned socket-based implementations and 3–8 compared to conventional systems. Webswamp can offer loads that are 2–8 times heavier. The TCP forwarder provides 50–300% higher throughput while also providing end-to-end TCP semantics that cannot be achieved with POSIX sockets. With more detailed measurements and profiling, these overall performance improvements are also broken down and attributed to the specific specializations described, providing server writers with insights into where to focus their optimization efforts.

Keywords: network services, extensible systems, OS structure, fast servers.

1 Introduction

Application-level networking allows regularly-privileged applications to interact (almost) directly with a network interface for their communication activities. This organization allows application writers to directly manipulate their communication patterns and protocol semantics, which can enable network communication systems to evolve more quickly and provide higher performance than monolithic kernel- or server-based networking. For example, the transactional TCP (T/TCP) protocol [8], which can provide higher HTTP performance, would undoubtedly have become commonplace years ago if support for it could be bundled with web browsers and servers. Application providers have more incentive to make their applications behave well than do operating system providers. In addition to enabling faster innovation, application-level networking also allows for improved performance. As we show in this paper, the ability to integrate and specialize networking code with application code can provide substantial performance increases for real applications (e.g., up to a factor of 8 for an HTTP server).

An inherent problem with conventional systems is that they allow only privileged servers and the kernel to interact with a network interface. Non-privileged applications are restricted to the interfaces and implementations of this privileged software. This organization can limit application functionality and performance. An interface designed to accommodate every application must anticipate all possible needs. The all-serving implementation of such an interface must resolve all tradeoffs and anticipate all ways that the interface could be used. Experience suggests that such anticipation is infeasible and that the cost of mistakes is high [2, 6, 10, 20, 25, 43].

More specifically, a network service's performance and functionality are often dictated by its network interactions, and yet support for demanding network-oriented applications remains primitive in most systems. Specifically, the network abstractions found in current OSs are generally high-level, inflexible and (most importantly) a poor match to the needs of many network applications. While appropriate abstractions can simplify the construction and improve the portability of applications, inappropriate abstractions often make it difficult or impossible to achieve semantic and performance goals. For example, software overheads and inefficient use of resources prevent many network data servers from exploiting the full performance of the underlying network. The avoidance of operating system latencies alone have been shown to provide significant benefits (e.g., sub-100-microsecond round-trip latencies) [49]. It has also been clearly demonstrated that servers can use hardware resources much more efficiently than current OSs allow [22, 26]. Further, because high-performance and correct semantics are critical, inappropriate abstractions increase complexity as application writers struggle to match incompatible needs and interfaces to recoup some of the desired behavior.

Application-level networking has the potential to eliminate these problems. By minimizing the software that cannot be bypassed to minimal primitives (e.g., open stream, send packet, receive packet) required for inter-application protection, one provides much greater flexibility. This flexibility will allow application writers to specialize their networking activities to application-specific needs, modifying default protocols and protocol implementations as necessary. Rather than requiring all applications and all systems to adopt the same new software, as current kernel-based approaches do, application-level networking allows distinct networking software to be bundled with different applications.

In this paper, we describe libraries and system daemons that safely and efficiently provide full, extensible application-level network services on top of the xok/ExOS exokernel system [25]. These services include UDP/IP, TCP/IP, POSIX sockets, ARP, DNS, and tcpdump. Our conventional networking abstractions provide performance that is competitive with a modern BSD system. In fact, the application-level library for POSIX sockets outperforms the kernel-resident BSD socket implementation by up to a factor of two. Simultaneously, on the same system, one can safely execute aggressively-specialized networking applications (e.g., a high performance HTTP server and a protocol forwarder with correct end-to-end semantics).

Our work builds on and augments earlier work in application-level networking in three main ways: (1) it provides concrete examples of exploiting the specializability of application-level networking to improve end-to-end performance substantially, (2) it demonstrates that and describes how full network services can be implemented as independent, application-level libraries, and (3) it identifies a set of base kernel mechanisms on which one can successfully do application-level networking.

The remainder of this paper is organized as follows. Section 2 discusses application-level networking in general, including previous work, basic design considerations, and the general exokernel system architecture. Section 3 details the design and implementation of xok/ExOS's networking components, including the kernel network interfaces and mechanisms, and the application-level implementation of standard network services. Section 4 describes the experimental setup used in Sections 5–7 to evaluate the efficiency of xok/ExOS's mechanisms and the benefits of their flexibility. Section 8 discusses interesting complications, lessons learned, and open questions. Section 9 summarizes the paper's contributions.

2 Background

In most systems, networking software is hidden away in the operating system kernel; only an abstract communication interface (e.g., POSIX sockets) is exposed to application writers. For some applications, this is an ideal arrangement, since the abstract interface is often a cross-platform standard that provides portability. However, when the interface is a poor match to an application's needs, the application writer is left with few options. Only those interactions explicitly supported by the kernel interface can be specified. Further, only behaviors anticipated by the kernel implementors will be supported well—for example, witness the order of magnitude performance problems observed in [4, 21].

We are certainly not the first to note this dilemma. Abstractly, the end-to-end argument addresses this dilemma—the Internet's protocol designers used end-to-end arguments to provide remarkable scalability and robustness. Operating system implementers violated the argument by embedding the resulting protocols in kernels, stopping just short of the true end-points: the applications. There have been a variety of approaches to addressing this dilemma, including application-level networking. The remainder of this section discusses this previous work, overviews the support needed for application-level networking, and briefly overviews the exokernel operating system architecture.

2.1 Related Work

This section discusses previous approaches to improving and advancing support for applications' varied network communication needs, broken into three categories: better kernel interfaces and implementations, extensible operating systems, and application-level networking.

Better kernel interfaces and implementations. The traditional system organization can be modified to better support new distributed applications with two types of enhancements. First, there have been many proposed improvements to the implementations of standard networking interfaces over the years. Some notable enhancements targeted specifically for emerging network services include Fbufs [13], LRP [12], and better implementations of the select system call [4]. A second, and ultimately more powerful, form of enhancement is to provide more suitable and more flexible interfaces. Perhaps unfortunate examples of this are the various "setsockopt", "fcntl" and "ioctl" calls for tweaking the behavior of network sockets in a small set of predefined ways. Many superior interfaces for asynchronous and copy-free data movement have been proposed [3, 23, 27, 36, 37, 39, 48, 49], though most systems are still not using them. An interface with great potential is NT's "send file" system call, under which several interesting performance enhancements could be implemented if the internal file system and TCP/IP implementations were re-organized.

Both forms of enhancement suffer from major pragmatic and fundamental limitations. The main pragmatic limitation is that it usually takes many years for a new OS interface or implementation technique to move from research prototypes to real systems, no matter how effective the research results show it to be. Part of the reason for this is that changes become more complicated as an OS matures, because enhancements often involve breaking modularity boundaries and adding "fast path" replicas of existing code paths. Even if OS implementors could move faster, a fundamental limitation is that support is always restricted to the needs foreseen by the implementors. Computer system evolution continues to show that OS implementors possess neither oracles nor omniscience.

Application-level networking offers a solution to both problems. By allowing application writers to bundle new implementations and interfaces with their applications, application-level networking bypasses the need for the critical shared resource (the OS kernel) to change for them.

Extensible operating systems. Several mechanisms have been proposed, evaluated, and deployed to allow applications to replace resource management abstractions with varying degrees of safety. For example, DOS and Alto [28] are completely open operating systems that give arbitrary applications direct hardware access but provide no protection. Other systems, such as Netware [31] and Windows NT, allow arbitrary flexibility for privileged users only, again with no protection. Of course, arbitrary static flexibility is also available to anyone with source code to an operating system and sufficient access to boot it on a machine; this source of flexibility forms the basis of Network Appliance's and Plan 9's approaches to building high-performance networked data servers [22, 40]. Several recent extensible operating systems, such as the Cache Kernel [10], Vino [43], SPIN [6, 19], and the exokernel [15, 25], focus on allowing similar degrees of flexibility for arbitrary applications while maintaining inter-application protection and fault-isolation.

All of these approaches to OS extension provide the flexibility needed to support changing needs. This paper describes how the networking support in the exokernel provides speed and flexibility.

Application-level networking. Several previous researchers have proposed, evaluated and developed mechanisms for supporting application-level networking (e.g., [34, 30, 47, 14, 49, 25, 7]). These efforts provided the arguments for the approach, showed that it could be used for some protocols, and demonstrated clear benefits for important workloads (e.g., cluster-based parallel computation). Based on this prior work, a standard network interface specification (called the VI Architecture [1]) is being developed and promoted by a set of companies led by Compaq, Intel, and Microsoft.

Most of the prior work on application-level networking has focused on the latency improvements provided by avoiding kernel entry and exit. A notable exception is the Nemesis system [7], which focuses on the quality of service benefits provided by removing networking from the shared kernel. (The Lazy Receiver Processing architecture [12] provides many of the same benefits for in-kernel implementations.)

We believe that the most important aspect of application-level networking is the control that it gives to application writers—network protocols and implementations can be integrated with and specialized for application activities, providing order of magnitude increases in performance. Our work extends earlier work in three main ways: (1) by providing concrete examples—such as the Cheetah HTTP server—of exploiting the specializability of application-level networking to improve end-to-end performance substantially, (2) by demonstrating that and describing how full network services including TCP/IP, UDP/IP, POSIX sockets, ARP, DNS, and tcpdump can be implemented as independent application-level libraries, and (3) by identifying a set of base network interface (NI) and OS mechanisms on which one can successfully do #1 and #2.

2.2 Application-Level Networking

With application-level networking, untrusted applications transmit, receive, and process their own network packets. This requires OS support for transferring packets between applications and a low-level network interface exported by either an advanced network interface card (NIC) or a device driver that emulates a NIC's role. This section briefly describes general application-level networking design issues; Section 3 describes in detail the xok/ExOS exokernel's implementation choices.

Overview. The central requirement for safe application-level networking is multiplexing the network interface. Multiplexing, or safely sharing, a network interface among a set of applications (rather than giving ownership to a single entity) involves two main sets of issues: those related to sending packets and those related to receiving packets. Common to both are some resource management issues for the host operating system, including notification, scheduling, and memory accessibility. Each of these design issues is discussed below.

Transmission. At one level, application-level transmission of packets is straight-forward—an application can simply give the network interface a pointer to the memory region(s) containing the packet to be sent. Assuming that the NI includes DMA support, it can pull the packet from memory and put it on the wire. For privacy purposes, there are also issues of pinning the memory regions and ensuring that they are actually readable by the transmitting application. Also, the application should be notified when the packet data has been copied from the memory regions—until this point, the application should avoid over-writing the regions in order to avoid corruption of the packet being sent. Two somewhat controversial design considerations center on whether and how to provide transmit queue fairness (among applications) and pre-transmission verification of packets.

Reception. The packet reception component of NI multiplexing can be broken into two steps: packet demultiplexing and packet buffering. *Packet demultiplexing* is the process of identifying with which connection or application a particular packet should be associated. It is the decision process by which the contents of incoming packets are kept private from all applications except those that have rights to them. Generally speaking, packet demultiplexing is accomplished by checking particular data offsets in the packet (usually corresponding to values in the various headers). One powerful approach to doing this is to use some form of *packet filter* technology [34], which involves pattern matching a pre-loaded set of <offset,value> pairs with the packet contents. A match identifies the appropriate receiver, and unmatched packets are handled via some default path (e.g., by dropping them or giving them to a default OS routine). *Packet buffering* involves safely delivering newly arrived packets to the applications for which they are destined. Generally, this involves copying the packets into pre-registered memory regions. If the NIC can do the demultiplexing, then it can place the packet into the correct buffer directly. Otherwise, a programmed copy is required. Safety requires that any physical pages directly accessed by a NIC be pinned to prevent page faults or, worse, re-allocation by other applications. Also, if there are no empty pre-registered buffers when a packet arrives, the packet is usually dropped. One alternate option is to buffer packets in the kernel (or on the NIC) and to give an application in-place access to their packets. However, this is complex with current memory

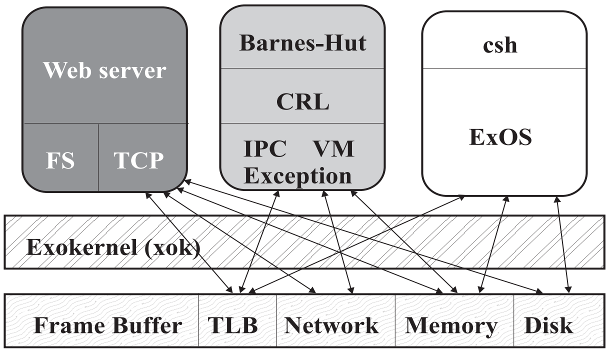


Figure 1: Exokernel Operating System Architecture. A small "exo"kernel enforces protection (multiplexing) of hardware resources, but otherwise avoids abstraction and non-hardware-protection functionality. Applications can link libraries to obtain desired abstractions, such as the POSIX-like library called "ExOS" used by the standard UNIX `csh` application shown. The web server and Barnes-Hut applications shown use more specialized libraries that contain only the specific functionality they require.

protection mechanisms.

Resource management. Notification, process scheduling, and memory management are important aspects of both transmission and reception, and many options exist for each. For example, notification approaches range between polling on status registers and interrupt-like schemes (e.g., upcalls or signals). Process scheduling can integrate some awareness of network events or be independent, with a corresponding impact on round-trip latencies. Memory management must deal with the need for pre-allocation and pinning, balanced against more conventional activities.

2.3 Exokernels and Library OSs

The work and experiments described in this paper is performed in the context of the Xok/ExOS exokernel system [25]. The exokernel operating system architecture [15], shown in Figure 1, was designed to give applications direct control over the management of their hardware resources while providing strong inter-application protection and fault-isolation. In this architecture, a small kernel, called an *exokernel*, does nothing except multiplex (i.e., time-share) the hardware resources among the applications running on the system. Applications are given direct, low-level access to their resources. The high-level resource abstractions that are traditionally provided by operating systems are instead provided in libraries against which applications can link. This organization allows application writers to select from among multiple resource management options or to construct their own. The exokernel ensures that aggressively specialized (and even buggy) resource management strategies and implementations can be used by critical applications without interfering with the correct operation of other programs on the system.

The ideal being pursued is that of safely allowing applications to bypass the high-level interfaces and complex implementations of conventional operating systems. This would allow applications to interact with their hardware resources without going through any intermediaries (kernel or server) and without looking through any abstractions. In practice, of course, inter-application protection has required that the exokernel insist on some degree of abstraction. Also, a server or two has crept into our exokernel system prototype when we attempt to precisely and securely emulate certain POSIX semantics. Still, by targeting the ideal case, we have learned a great deal about how to support and build application-level services, some of which is discussed in this paper.

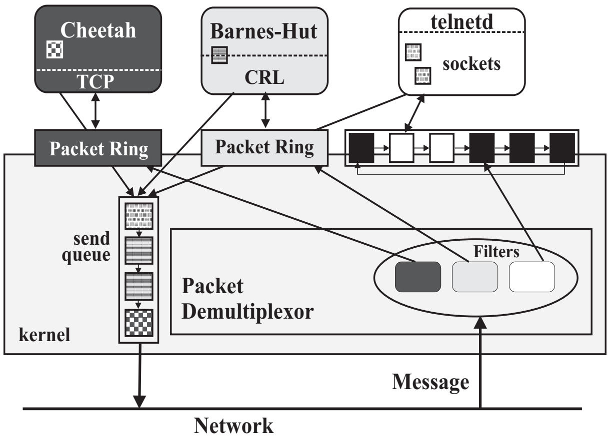


Figure 2: xok/ExOS's Application-level Networking Architecture. Applications transmit and receive packets "directly" via virtual network interfaces. Received packets are compared to pre-specified packet filters and copied into a pre-registered buffer associated with the matched filter. Transmitted packets are linked into a shared send queue without copying the data; the four small squares in the application regions contain the data for the four packets currently in the send queue. Three applications are shown: the Cheetah HTTP server, which will be described further in Section 5, the Barnes-Hut parallel application on top of the C Region Library (an explicit distributed shared memory system [24]), and the BSD TELNETD application on top of a POSIX-compliant TCP/IP socket library.

Xok is an exokernel for Intel x86-based computers, and ExOS is its default, POSIX-like library operating system (libOS). Xok/ExOS is self-hosting (i.e., it can be edited, compiled, linked and rebooted on itself), and can run many unmodified UNIX programs (e.g., perl, gcc, telnet, and most file utilities). Relevant to this paper, it also supports a wide range of standard networking protocols, tools and applications, including UDP/IP, TCP/IP, ARP, DNS, NFS, FTP, FTPD, POSIX-like sockets, and a tightly-integrated, highly-specialized, very efficient HTTP server. The remainder of this paper details how.

3 Application-level Networking on an Exokernel

This section describes the application-level network services implemented in the xok/ExOS exokernel system [25]. It includes a high-level description of the application-level networking architecture and detailed descriptions of xok's network interface multiplexing mechanisms, other relevant xok mechanisms (e.g., for memory mapping and scheduling), and ExOS's implementations of basic network services.

3.1 Overview of Software Architecture

Figure 2 shows the basic architecture of xok/ExOS's networking system, in which applications interact (almost) directly with the network interface, a small kernel component makes this safe, and libraries provide applications with networking abstractions that correspond to their needs. Outbound packets are sent directly from application memory. Applications invoke the "send_packet" system call to add descriptors of outbound packets to the kernel's FIFO send queue. Inbound packets are received by xok's network device drivers (linked into the kernel) and passed into the kernel proper. xok identifies each packet's destination by use of a packet filter engine and then

Transmit	net_xmit	Asynchronously transmit a packet on a given network interface
Demux	dpf_insert	Insert a filter and associate a packet ring with it
	dpf_delete	Dereference a filter
	dpf_ref	Add a reference a filter for a new process
Buffering	dpf_pktring	Route filter's matches to a new packet ring
	pktring_setring	Set up a new packet ring with specified set of entries
	pktring_modring	Add, delete, or replace packet ring entries
	pktring_delring	Delete packet ring from kernel's view
Others	wkpred	Install wakeup predicate
	insert_pte	Insert page table entry

Table 1: The main kernel functions for supporting application-level networking.

copies it into one or more rings of pre-registered buffers shared between the kernel and applications. The remainder of this section explains both the kernel mechanisms and the application-level implementation of standard network services.

3.2 xok Interfaces and Mechanisms

This subsection describes the components of xok that are directly relevant to supporting fast and flexible application-level networking. Most of these were outlined in [25], but they are revisited here with a focus on how they relate to application-level networking. The abstractions used by xok to make inter-application protection tractable are the only ones that cannot be bypassed by application writers. Therefore, their proper design will be critical to avoiding limitations down the road. Fundamentally, our NI is quite similar to previous user-level NIs; there are some interesting differences, but we describe xok's mainly to provide context for the application-level network services described subsequently. Also, xok's non-NI support is significantly different from that of other systems. Xok's main kernel functions for supporting application-level networking are listed in Table 1. We discuss them in detail.

Memory Management. xok manages three main data structures to multiplex main memory: per-process x86-defined virtual memory (VM) page tables, per-page structures that identify which processes have which access rights, and a free list. Physical pages are free when no processes have access. Access rights are added in two situations: when a process allocates a physical page from the free list and when a process with access gives the same or lesser access to another process. Although it is outside the scope of this paper, this base support allows for hierarchical memory management. A process can map its own page tables read-only. Processes can also direct xok to modify their page tables by removing VM mappings or adding mappings to pages to which they have appropriate access. Because each process manages its own page tables, processes can ensure that they have physical pages backing important virtual address ranges, and they can share memory with other processes in arbitrary and dynamic ways. When it is necessary to revoke an allocation from a process, the kernel makes an upcall to that process—the process can then pick a page, save it if needed and give it back to the kernel.

xok always "pins" physical pages that are being shared by the kernel, a device driver, or a device. For example, this includes pages holding data to be transmitted and pages that hold notification variables. In this context, we are using the term, "pin", to indicate that the kernel will not allow the page to be allocated from the free list, even if it were to be made free. This does not imply that the kernel makes the page read-only or unaccessible, and the kernel does not prevent the application from unmapping the page.

Efficient Polling. WK (or WaKeup) predicates provide applications running on xok with a general mechanism for very efficient polling. A WK predicate is a set of conditions (in sum-of-products style) that describe the circumstances of interest to an application. Each condition consists of comparing some memory location to either a constant value or another memory location. When a process wants to wait for certain circumstances, it constructs the appropriate predicate and gives it to xok. xok checks and dynamically compiles the predicate into efficient native code; xok also pins all pages accessed by the predicate. The process scheduler will always evaluate a sleeping process's current predicate (if any) before it considers giving the CPU to that process. If the predicate evaluates to true, the process is marked runnable. To date, all of the circumstances for which we have used polling (e.g., *sleep()*, *select()*, IPC, and I/O event notifications) have been expressable as simple predicates. As a result, many fewer context switches are necessary as the system becomes populated with many polling processes.

CPU Scheduling. The current xok CPU scheduler manages the CPU with no particular consideration for I/O devices. The one exception to this rule is that the current scheduling decision is reconsidered whenever an I/O event (e.g., a packet arrival) related to a sleeping process occurs. This reconsideration includes evaluating any corresponding WK predicates. If the I/O event causes the corresponding process to awaken and that process should be granted the current CPU time, the newly awakened process will be scheduled. I/O events that relate to a runnable or running process do not cause the scheduler to be invoked. Also, the fact that the process is made runnable does not guarantee that a context switch will occur immediately.

Packet Transmission. xok's packet transmission interface is quite simple, essentially consisting of a system call that takes three parameters: the interface on which to send the packet, an array of <address,size> pairs, and a pointer to an integer in memory. The kernel makes certain that the initiator has read access to the set of memory regions and write access to the integer. Then, unless the relevant device's transmit queue is full, the various pages are pinned and the packet descriptor is appended to the queue. When the device driver eventually learns that the packet has been sent by the card, xok decrements the integer in memory and unpins the pages. Decrementing the notification integer fits well with the WK mechanism and allows both per-packet and grouped handling by applications. It is up to the application to avoid over-writing its own packet until it has been sent, which can be inferred from the value of the integer. xok's current implementation enforces no inter-application fairness with respect to the transmit queue and allows applications to transmit any data desired onto the network.

Packet Demultiplexing. xok uses a packet filter engine, called Dynamic Packet Filter (DPF) [16], to demultiplex packets among receivers. DPF uses dynamic code generation to make packet filtering much more efficient than previous software packet filter systems [34, 42]. The downside of DPF is that filter installation is somewhat more expensive. In other respects, it is similar to these previous packet filter systems. Applications request access to particular types of received packets (e.g., TCP packets to port 80) by constructing the corresponding DPF filter and passing it to the xok "install_filter" system call. The kernel verifies that the new filter either does not overlap an existing filter or that the application has access to said filter.

Packet Buffering. Associated with each filter is a *packet ring* identifier. A packet ring is a ring of buffers set up by an application and shared with xok. Each buffer has a field that indicates the owner: xok if zero and not-xok otherwise. As shown in figure 2, when a received packet matches a filter associated with a particular packet ring, xok tries to copy it into the current packet ring entry. If the entry is not owned by xok, the packet is dropped. If the packet is longer than the entry, it is truncated. After the contents have been copied into place, the original packet size is written into the ownership flag, simultaneously giving notification and the size information to the application. Applications handle their packets when and how they choose; they return buffers to the kernel by simply writing a zero to the ownership flag. This interface involves one copy for each received packet, because the kernel performs the demultiplexing; this copy can be avoided with NICs that perform the demultiplexing.

3.3 ExOS Networking Abstractions

This subsection describes, with specific examples, the key mechanisms used in ExOS to achieve efficient application-level network services. In the process, we will attempt to point out places where, in retrospect, we could have made better design choices. Because most of ExOS is just a library linked by applications, like the math library, our descriptions will often use "ExOS" and "application" interchangeably when describing how ExOS works. It is important to note that aggressive applications, like the Cheetah HTTP server described in Section 5, can bypass any and all aspects of the ExOS support by simply not invoking them.

UDP/IP. The User Datagram Protocol (UDP) is a simple protocol, and it therefore represents a good example to start with. UDP is connectionless and has few built-in protocol activities (no retransmission, no flow control, etc.). To create any networking end-point on xok, an application (1) allocates physical memory, constructs a packet ring, and registers it with xok, and (2) constructs a DPF filter description and registers it with xok specifying that it be associated with the new packet ring. After this setup, xok copies incoming packets that match the new filter into the new packet ring. To create a UDP end-point, ExOS builds a DPF filter to identify Internet Protocol (IP) packets that specify UDP as the protocol and have the desired source and destination IP addresses and UDP port number values. The source IP address, the destination IP address and the source UDP port number can be left as wildcards, meaning that they are not checked during DPF's pattern match for this end-point's filter. Any combination of these values that does not overlap existing filters will be allowed.

Once an application has made itself an end-point, it will receive packets and can process them when and how it likes. To sleep waiting for packets, the application constructs a WK predicate that tells the process scheduler to leave it sleeping until the ownership flag of the next packet ring entry changes from zero (i.e., owned by xok) to non-zero. When a packet arrives, xok copies it into the packet ring and puts its non-zero size into the ownership flag. The packet, in UDP format, can then be parsed and processed by the application. To send a UDP packet, an application constructs an appropriate header and passes to the xmit_packet system call a gather list of pointers to the header and the packet contents. The application will know that it can re-use the memory regions holding the header and data when the notification integer specified in the xmit_packet call is decremented. (Note: no packet rings or DPF filters are required for packet transmission.)

TCP/IP. Operationally, application-level Transmission Control Protocol (TCP) end-points are established and used in the same way as UDP end-points. However, TCP is a more involved protocol and requires additional support from the underlying system in order to function effectively and correctly. Some aspects, such as generating acknowledgements and holding onto transmitted data buffers until acknowledgement rather than letting them be re-used after transmission, are just part of the TCP implementation. Other aspects involve kernel mechanism or careful organization of the core ExOS code. The four main examples are:

1. Listen/accept. TCP is a connection-oriented protocol, which means that two end-points exchange messages to establish a connection before data are actually exchanged. The common approach to doing this is to have one side establish an end-point that "listens" at a particular TCP

port number for other end-points to send a "connect" message. The listener then acknowledges the connect message with a connect message of its own, thereby "accepting" the connection. Generally, the new connection is viewed as a new end-point on the listener's system and the listen end-point remains intact. To establish this new end-point, an application can create a new packet ring and a new DPF filter that gets the correct subset of the listener's packets. DPF allows the creation of this more specific *subsetting* filter, so long as the creator has the proper capability for the filter being subsetted. For performance purposes, ExOS generally does not actually create a distinct xok-visible end-point until it is required for inter-application protection, if it ever is. Instead, the listen end-point continues to get the packets for both the listen end-point and the new connection. A second demultiplexing step in user space identifies the specific connection. This decision is discussed further below.

2. Timers and Timeouts. TCP requires timers[1] to trigger a number of activities. For example, TCP is a *reliable protocol*, which means that it retransmits data that are not acknowledged and therefore may have been lost. To do this robustly, TCP waits some period of time before trying again. It is straightforward to have the TCP code check for timeouts and arrived packets when it is executing. However, an application may compute and/or sleep for extended periods of time between calls into the TCP code. Therefore, ExOS supports *WK add-ons* and *context switch add-ons* to enable timeouts and arrived packets to be handled in a timely manner. These add-ons allow an application library to register functions to be called by ExOS each time a context switch upcall is being processed or a WK predicate is being downloaded in the given application. The application's context switch add-ons are called by ExOS when the process is given the processor for a quantum.[2] ExOS's TCP code uses a context switch add-on to check for and process timeouts and arrived packets "in the background". This works well for TCP's purposes, because the timers can be coarse-grain and imprecise without affecting correctness—and when performance is critical because a lot of traffic is going through the TCP code, the internal timer checks provide greater precision. WK add-ons are called by ExOS when a WK predicate is to be downloaded, allowing application modules to add "OR" conditions to the predicate. ExOS's TCP code uses the WK predicate add-on to cause the process to be awakened at least by the next timeout or packet arrival. A handler function associated with the WK add-on processes the TCP timeout, if that is the reason for waking up. After calling such a handler, ExOS reconstructs the WK predicate and puts the process back to sleep via the same algorithm as before.

3. TIME_WAIT. TCP includes explicit support for preventing packets of previous connections from being processed as part of a new connection. It does this by having at least one end-point of a closing connection enter and remain in the TIME_WAIT state for two times the maximum round-trip time (specified as two minutes in the TCP specification [41]). This presents a problem for application-level TCP implementations when applications want to close their connections and terminate. The easiest solution, which ExOS used for awhile, is to have the process wait for all of its TIME_WAIT connections to fully close before allowing the process to terminate. Unfortunately, this approach can have adverse effects on performance for applications, such as command line shells, that wait for child processes to complete before continuing. The revised

[1] We distinguish here between passive use of timers to measure delays (e.g., the round-trip delay) and active use of timers to trigger TCP-related activity. It is the latter that we are discussing here. To assist with the former, we have considered having xok provide the packet arrival time in addition to the packet length when it fills packet ring entries. This would make round-trip time measurements independent of delays between packet arrival and packet processing.

[2] Recall that exokernel applications participate in their own context switching (e.g., to save and restore registers). This activity, which is a core component of the ExOS library, can also be specialized for other purposes, such as the add-ons discussed here and software interrupt masking (for application-level critical sections on uniprocessors).

solution exploits the completely application-level nature of ExOS to deliver the "child is done" signal to the parent process before actually terminating. Thus, a process can effectively terminate from the POSIX process hierarchy point of view, while persisting until all TIME_WAIT TCP connections close. A third approach, which we recommend for most other types of systems, would be to simply hand-off TIME_WAIT connections to a special-purpose server when the process wants to terminate. This hand-off involves a simple sequence of steps: provide the server with the few relevant TCP control block values, give the corresponding DPF filter to the server, and switch its association from its current packet ring to the server's packet ring.

4. Port Number Selection. Implicit port number selection is an important aspect of a TCP implementation. Although an application can choose its own if it desires, often it asks the system to pick a good one for it. A "good one" means, at the least, one that is not currently in use. Because of the TCP TIME_WAIT state, it is also preferable to select one that has not recently been used, since the application may try to use the same service on the same server as the previous user of that port number. Our current approach, provided by our desire for maximum decentralization, is to pick a random port number in the allowed range (e.g., 1024–65535) and repick if the DPF filter is rejected because someone else is currently using that port number. This has worked well for us in practice. However, it will occasionally select a recently used port number, which DPF will not complain about, and use it to open a connection to the same server as before. In this case, a process can wait for a lengthy period of time for the remote TIME_WAIT state to clear. For this reason, centralized TCP implementations generally sequence through the valid port number range to reduce the probability of such a collision. It is still possible (e.g., due to explicit port number selection and long-lived connections), but much less likely. We are considering the use of a simple protected method (i.e., a safety-checked system call) to provide this same support.

When all of the above is combined, what we have described and built is a completely-decentralized, application-level TCP implementation. It has all of the performance benefits and flexibilities of application-level networking, and it communicates correctly with every TCP system that we have tried. In addition, because it is at the application level, we have found it to be much easier to enhance its performance. For example, by minimizing and isolating protocol control blocks for TIME_WAIT connections, we have largely eliminated the "TIME_WAIT problems" reported in [33, 17]. As a result of this and other modifications (e.g., heavy use of precomputed values and careful recycling of buffers), our application-level TCP implementation outperforms that of a popular BSD TCP by up to a factor of two (see Section 5).

POSIX Sockets. POSIX sockets provide a level of abstraction on top of UDP/IP and TCP/IP end-points (as well as other protocols). Therefore, they have all of the same basic requirements and components as described above for the protocols themselves. In addition, they provide a number of additional features to hide characteristics of the underlying protocol. For example, sockets process packets in the background and buffer received data if the process has not already provided a destination memory region. Also, sockets buffer outgoing data and, for TCP, keep the buffered data until it is acknowledged for retransmission purposes. All synchronous socket calls (i.e., calls that may involve waiting for network packets) use WK predicates as described earlier. This is also true of the *select* call, which waits for activity on any of a set of sockets. It is important to note, however, that the application process may awaken to do work in the socket library but then go back to sleep immediately after because the work only moved it part way towards completion of the original socket call.

Inter-Endpoint Resource Sharing. It is fairly common for network-intensive applications to simultaneously utilize multiple network end-points. It is important, in these cases, to share resources such as buffers among these connections rather than statically partitioning such resources

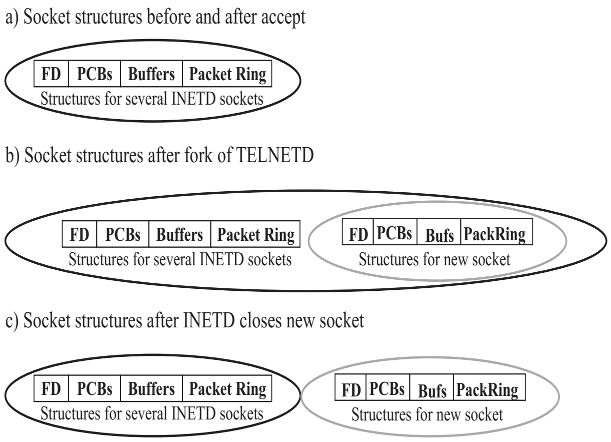


Figure 3: Sharing of socket structures in `inetd`. This example shows the sharing of connections between `inetd` and a child `telnetd` process as `inetd` (a) accepts a new connection, (b) forks the `telnetd` process for it, and (c) closes the handed-off socket. Each connected set of boxes represents a set of related physical pages, and the ovals indicate which processes have access to them. The black oval represents `inetd` and the grey oval represents `telnetd`.

among them. This sharing can also help to eliminate end-point setup costs for applications that involve many simultaneous or short-lived connections. ExOS accomplishes this sharing across endpoints owned by a particular application in several ways. First, as discussed above for TCP, we exploit the fact that the DPF filter for an *accept*ed connection is a strict subset of the filter for the *listen* end-point. This allows us to avoid any xok-level setup for the new end-point, replacing it with ExOS-level demultiplexing across end-points owned by the application. Second, we exploit the fact that xok allows multiple DPF filters to place their packets into a single packet ring. When a new end-point is desired for *listen* or *connect*, the relevant DPF filter is registered with xok and associated with the process's main packet ring. Third, we exploit the fact that socket-level buffers are completely in application space to share the pool across sockets owned by that application. These mechanisms reduce the CPU time and memory space requirements of applications, like network servers, that use multiple connections.

Inter-Process Socket Sharing. An important aspect of POSIX sockets is inter-process socket sharing, which generally occurs when one process opens a socket and then forks off a child process; both then share the open socket. The key to supporting this with application-level networking is ensuring that everything associated with a set of sockets can be encapsulated in a specific set of pages. This includes the file descriptor and socket structures, the protocol control blocks, the socket data buffers, and the packet rings exposed to the kernel for received packets. To simultaneously support the inter-endpoint sharing just described, the ExOS software and structures are organized to allow extraction of a socket from one such a set into a distinct set of pages. Also, a single process can simultaneously utilize several such sets. This allows ExOS to dynamically isolate one set of sockets from another. This ability allows ExOS to enforce a variety of protection models, ranging from share-it-all-with-everyone to use-a-server-whenever-there-is-more-than-one-app-with-access. Our default is in the middle—socket state for any given socket is shared by exactly that set of processes that have access to it. Full support of POSIX's socket isolation semantics would require the use of a third-party server (as in micro-kernel systems) for policing access to shared socket structures.

Figure 3 illustrates how ExOS uses the ability to dynamically separate sets of socket structures into distinct sets of pages. Specifically, it shows the `inetd` network service accepting a network connection, forking a `telnetd` for that one new connection, and then closing the one new connection. Before and after the accept, the ExOS structures related to all of `inetd`'s connections share one set of pages. In the process of forking the new process, ExOS separates the one shared connection into a distinct set of pages, which it maps into the child process's address space at the same virtual addresses. When `inetd` subsequently closes the socket, it unmaps the pages, completing the socket hand-off.

System-Wide Services. There are a number of system-wide services that are integral to the successful provision of true networking services. This section describes three of these to illustrate how such services are provided at the application level on top of xok's interfaces.

1. Naming and Routing. Naming, or the translation of a "higher-level" identification to a "lower-level" identification, is core to networking. Two prime examples are the Domain Name System (DNS), which translates alphanumeric machine names to IP addresses, and the Address Resolution Protocol (ARP), which translates IP addresses to link-level addresses (e.g., ethernet). Related to ARP-level translation is routing, whereby the link-level address for a non-local IP address is replaced by the link-level address of the first router or gateway along the path to the non-local machine. The key practical aspect of these naming services, from an implementors point of view, is safe management of the translation tables. The sharing model is that everyone can look at them, but updates need to be well-formed and correct according to the naming protocols' rules. (DNS and ARP both rely on wire trust and are therefore subject to spoofing attacks). Using ARP as a representative example, ExOS supports this sharing model by giving ownership of the ARP table to an *arpd* application. This application, started during system initialization, installs a DPF filter that gives it all incoming ARP packets, including ARP responses. The ARP table, which only *arpd* can update, is read-shared and mapped by all processes. When a process desires ARP information, it simply looks in the table. If the desired entry is not present, the application constructs and transmits an ARP request packet to get the information from other network nodes. All received ARP packets are delivered by xok to *arpd*, which updates the ARP table appropriately. An application that needs the update can use a WK predicate to wait for the table to be modified and repeat its table lookup. Of course, the application should also include a timeout clause in its WK predicate and repeat the ARP request, in case the ARP packet was lost. The same basic approach can be used for each of the name translation tables.

2. Error Reporting. An important practical consideration in networking is identifying erroneous packets and delivering notification of problems to end-points. The best example of this is the Internet Control Message Protocol (ICMP). Two aspects of this protocol are of special interest here: transmitting ICMP messages and receiving ICMP messages. When a valid IP packet with no interested end-point is received, an ICMP packet should usually be sent to the IP packet's sender with an indication of the problem (e.g., unsupported protocol or unreachable port). This service is provided in xok/ExOS by an ICMP server, started at system initiation time, that downloads a DPF filter that matches any IP packet but can be subsetted by any other application. In this way, any unclaimed IP packets will be seen and responded to by the ICMP server.

3. Monitoring of Traffic. Although it is not a core network service, it is often useful to be able to observe the traffic on the local network link. One common approach to doing this is exemplified by the UNIX *tcpdump* utility. *tcpdump* directs the kernel to give it copies of all received packets, which it then prints in readable formats. xok provides support for delivering copies of specific sets of packets to applications via a mechanism referred to as "copy filters". A copy filter uses the same DPF mechanism as regular filters, except that each copy filter is evaluated independently and filter overlapping is allowed. Because the arbitrary overlapping violates privacy, super-user

privileges are required to install a copy filter. As with regular filters, matched packets are copied into the associated packet ring. xok/ExOS's *tcpdump*-like application simply downloads a filter that matches all packets and processes the observed packets as specified in the command-line options.

3.4 Summary

The ExOS libraries and daemons described above safely provide complete networking services entirely at the application level. These services include UDP/IP, TCP/IP, POSIX sockets, ARP, DNS, and network monitoring. This is made possible by the network interface multiplexing mechanisms in xok and careful organization of core ExOS modules.

4 Experimental Apparatus

Sections 5–7 evaluate the efficiency and flexibility of xok/ExOS's application-level networking support. This section describes the experimental setup used.

All experiments were performed using standard personal computer technologies. All systems used include 200 MHz Intel Pentium Pro processors, the Intel VS440FX PCI chip set, and 64 MB of main memory. The systems all use SMC EtherPower 10/100 fast ethernet (i.e., 100Mbit/second) cards [44] and are connected via a dedicated SMC TigerSwitch 100 [45] high-performance fast ethernet switch. The cards are based on the DEC 21140A LAN controller chip [11] (a.k.a., the Tulip chip). The switch can interconnect eight full-duplex fast ethernets at nearly full speed on its 1.8 Gbit/second backplane. It also dynamically selects between cut-through and store-and-forward routing, to trade-off propagation latency and collision rates. One of the systems (used as the server when relevant) is equipped with three fast ethernet cards, and each client communicates via the switch to a separate card.

Most of our experiments focus on the xok/ExOS system. However, we also compare the end-to-end performance of our network service implementations to the performance provided by OpenBSD 2.2, which is a free operating system based on 4.4 BSD [32]. The xok device drivers were taken from OpenBSD, removing this potential variable. In general, OpenBSD outperforms the exokernel-based system used in these experiments, which is functional but has not been tuned significantly. ExOS's TCP and socket libraries, on the other hand, have been tuned and they consistently outperform OpenBSD when emulating the same interfaces—by 50–100% for the examples in this paper. This fact allows us to focus on the benefits of specializability, rather than having the result be clouded by unknown variables.

The performance measurements come from several sources. Measurements of elapsed time use the Pentium Pro cycle counter to get 5 nanosecond granularity on the 200 MHz systems. Network transmissions, disk requests, interrupts and other such event counts are maintained by the kernel and exposed read-only to applications. All measurements presented are averages of at least 5 executions, with coefficients of variation of less than 0.02 unless otherwise stated.

5 Cheetah: a fast HTTP server

This section illustrates the efficiency and flexibility of xok/ExOS's application-level networking services, using HTTP servers as a case study. First, the same HTTP server is shown to deliver twice the throughput when using the TCP/IP socket libraries described in Section 3 rather than when running on OpenBSD. Then, the Cheetah HTTP server is used to demonstrate one of the main advantages of application-level networking: specialization. By exploiting the ability to safely specialize application-level implementations, we demonstrate an equivalently functional HTTP server that delivers 3–8 times the throughput of servers running on OpenBSD. A brief synopsis of this section appeared in an earlier paper [25]. Here, we motivate, describe and analyze Cheetah's use of application-level networking.

5.1 Performance and Complexity Problems in HTTP Servers

It is quite easy to construct a functional HTTP server with the abstractions provided by current general-purpose OSs. With a few hundred lines of code, an application can listen for TCP connections via a socket interface, read requested documents from the file system, and write them to accepted connections. There are a variety of extensions (e.g., MIME header options and CGI scripts) to this simple model, but it captures the performance-critical core.

Unfortunately, the performance of such implementations tends to be poor for several reasons. First, all applications (HTTP servers and otherwise) on a given system share a single implementation of TCP sockets. Unfortunately, researchers have found the implementations present in most systems to be inadequate for HTTP servers [4, 33, 46]. The restrictive interfaces of current systems prevent application programmers from customizing the behavior of the implementation to avoid these performance problems. Second, the blocking nature of most file system interfaces and some socket interfaces restricts I/O parallelism. Third, the use of distinct, non-integrated subsystems (in this case, the file system and the networking stack) results in significant redundancy (most notably, repeated data copying), both in work and in memory usage.

Because HTTP server performance is critical in many environments, application programmers often adopt complex workarounds to recoup some of the lost performance, making these applications difficult to construct, debug and maintain. For example, to exploit I/O parallelism despite blocking I/O interfaces, early HTTP servers created a new process for each client request [29, 35]. To avoid the associated fork/exit overheads, more recent implementations maintain a pool of server processes [35], coordinating work via shared memory and interprocess communication. To avoid file system overheads and some copying from kernel space to application space, some applications replicate document caching functionality at user level [9]. Because HTTP server implementors do not have sufficient control over TCP implementations via current socket interfaces, complex connection reuse semantics (referred to as "Keep-Alive" or persistent connections) have been made a required part of the HTTP/1.1 protocol specification [18]. This complicates HTTP servers, requiring them to replicate portions of the OS networking code, and can lead to conflicts between user-level connection management and kernel-level protocol implementations (e.g., the order of magnitude performance decreases observed in [21]).

Clearly, server developers are willing to work to improve performance. Extensible systems allow them to do so more directly. With appropriately re-usable libraries, this does not have to significantly increase complexity. In fact, the Cheetah server described below is substantially faster and no more complex than the aggressively-optimized HTTP server using the standard interfaces.

5.2 ExOS Socket Performance

To evaluate xok/ExOS's default application-level network services, we constructed a socket-based HTTP/1.0 server. This socket-based server consists of one single-threaded application that aggressively uses non-blocking socket I/O for its TCP/IP networking in order to achieve good performance with standard interfaces. The file system interfaces are blocking, but the experi-

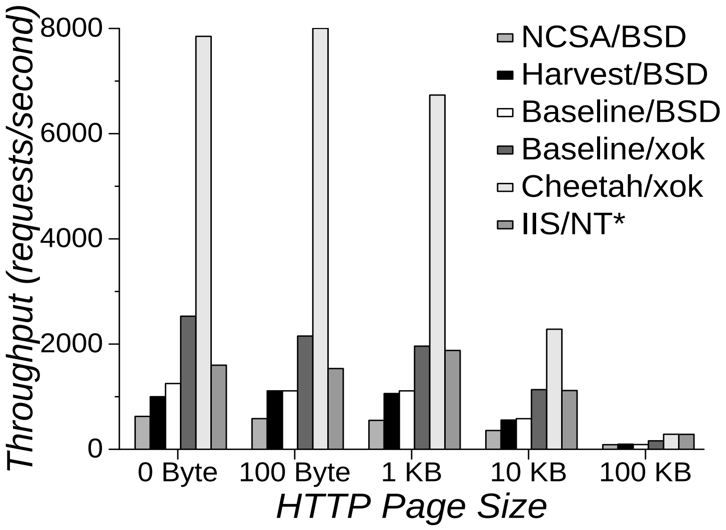


Figure 4: HTTP request throughput as a function of the document size for several HTTP/1.0 servers. "NCSA/BSD" represents the NCSA/1.4.2 server running on OpenBSD. "Harvest/BSD" represents the Harvest proxy cache running on OpenBSD. "Baseline/BSD" represents our socket-based HTTP server running on OpenBSD. "Baseline/xok" represents our socket-based HTTP server running on xok/ExOS. "Cheetah/xok" represents the Cheetah HTTP server running on xok/ExOS. "IIS/NT*" represents Microsoft's IIS server (version 2) running on Microsoft Windows NT. The "*" highlights the fact that these numbers are for a 300 MHz Pentium II system, as opposed to the 200 MHz Pentium Pro used for the others. The throughput values are obtained with 6 clients (2 per client machine) repeatedly requesting the same static web document (100% cache hits) with zero think time between the completion of one request and the initiation of the next. Care was taken to avoid performance degradation of the OpenBSD servers due to "TIME_WAIT" connections (by using spaced bursts of requests)—no such provision is necessary for servers running on xok/ExOS.

ments focus on file cache hit performance, removing this as an issue. For robust performance, additional complexity would be needed in this baseline server.

Figure 4 shows HTTP request throughput as a function of the requested document size for four servers: our socket-based server (referred to as "Baseline") running on OpenBSD, Baseline running on xok/ExOS, the NCSA 1.4.2 server [35] running on OpenBSD, the Harvest cache [9] running on OpenBSD, the Cheetah server running on xok/ExOS, and Microsoft's IIS server (version 2) running on Windows NT. (We also measured Apache 1.2.6 and 1.3 on both OpenBSD and linux, but their "out-of-the-box" performance was always lower than NCSA.) All requests hit in the server's file cache. Comparing the two Baseline servers allows us to evaluate xok/ExOS's TCP/IP socket implementations, while the NCSA and Harvest numbers give us confidence that the socket-based server's behavior is reasonable. The IIS numbers provide insight from another system and are particular interesting because of Windows NT's "sendfile" interface.

Each HTTP request consists of opening a TCP connection, receiving and parsing an ASCII request, returning an ASCII header and the requested file data, and closing the connection. Therefore, small HTTP requests exercise connection creation/termination performance, and large HTTP requests exercise bulk data movement performance. Figure 4 covers a spectrum that includes both, and provides several interesting pieces of information.

First, our base HTTP server performs roughly as well as the Harvest cache, which has been shown to outperform many other HTTP server implementations on general-purpose operating systems. Both outperform the NCSA server. This gives us a reasonable starting point for evaluating HTTP server performance.

Second, xok/ExOS's default socket implementation outperforms the OpenBSD socket implementation by 100% for small requests and 80% for large documents. This indicates that xok/ExOS's socket performance is quite good for both creation/termination activities and bulk data movement. We do not have detailed explanations for why OpenBSD's TCP socket performance is so much lower, though we do know that ExOS's implementation has been tuned and streamlined aggressively. However, none of the performance enhancements are specific to an application-level implementation, and there is no reason to believe that OpenBSD could not be made to match them. This allows us to safely run both our baseline servers and our specialized servers on xok/ExOS, removing several otherwise free variables.

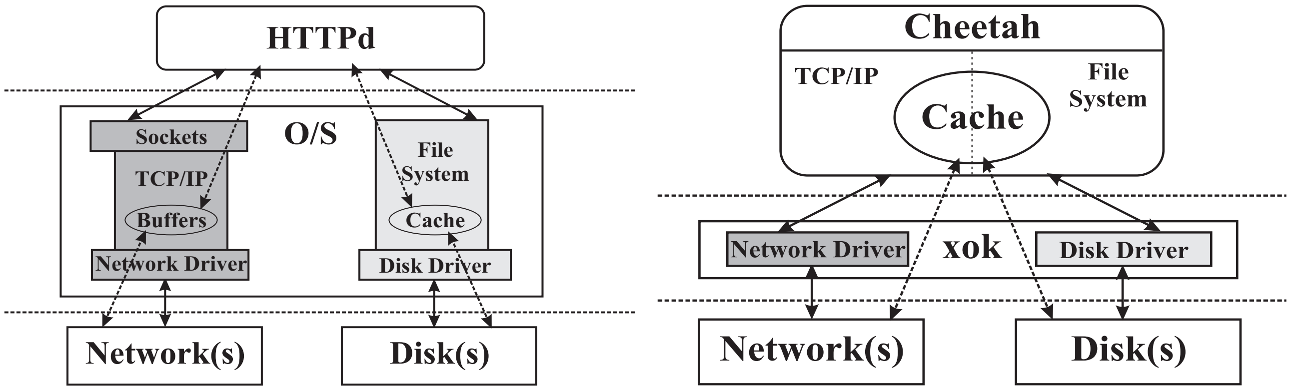
Third, the IIS/NT measurements indicate that, like the Baseline servers, it is limited by software overheads rather than the network for most file sizes. IIS/NT's advantage over the OpenBSD servers roughly matches the increased CPU performance (300 MHz versus 200 MHz) for the 100byte and 0KB document sizes. For the larger sizes, the advantage is larger, presumably because of the "send file" support. However, it's performance for 1KB and 10KB document sizes is still below xok/ExOS's sockets. For 100KB, IIS/NT is limited by the available network bandwidth (3 100Mbit/second Ethernets).

5.3 Exploiting application-level networking

The real power of application-level networking is that it allows networking to be specialized for and integrated with important applications. To provide a concrete example, this subsection describes the use of and quantifies the impact of such specialization in the context of an HTTP server, called Cheetah. Cheetah incorporates a number of performance enhancements enabled by application-level networking. As shown in Figure 4, these enhancements improve performance by a factor of 4 for small HTTP documents and 80–100% for large documents, when compared to the Baseline socket-based server (on xok/ExOS) evaluated in the previous section.

Cheetah uses much of the same code as the Baseline server, including all HTTP-related functionality (e.g., request translation, MIME header processing, response header generation, etc.). However, the main control loop is different. Cheetah simply waits for any relevant event (e.g., network packet arrival) and then processes it. For event processing, Cheetah calls into private, specialized instances of TCP/IP and file system libraries. Once an event has been processed, per-request state is checked to see if forward progress can be made on the HTTP request (e.g., the request has arrived and can be processed, or data is ready to be sent). This simpler and more direct implementation (relative to socket interfaces) of a non-blocking event loop provides some performance benefits, including automatically avoiding the *select* problems discussed in [4]. Avoiding layers of general-purpose code for file descriptors and sockets further reduces overhead. However, the largest improvements come from three extensions enabled by application-level networking:

Merged File Cache and Retransmission Pool: Cheetah avoids all in-memory data copying and the need for a distinct TCP retransmission pool by transmitting file data directly from the file cache. Specifically, it calls a low-level file system routine to obtain read-only pointers to file cache blocks, pin them and mark them copy-on-write. These pointers are then passed to a TCP library routine to construct packets that include data directly from the file cache. These packets



(a) Conventional HTTP server (b) Integrated HTTP server

Figure 5: HTTP servers on conventional systems and exokernel systems. On a conventional OS, many software layers and data copies are involved with an HTTP server's work. The solid lines represent control flow, and the dotted lines represent data copies. Application-level networking and exokernels allow the server to avoid these redundancies, integrating and streamlining the required software and merging the data repositories.

can be transmitted, and retransmitted if necessary, via xok's transmit interface. This provides zero-copy disk to network transfer, where the "to network" part is via a reliable protocol and can be repeated endlessly. One issue of potential concern is synchronization with other applications using the same files — Cheetah handles this by simply not sharing the files actively. Instead, the files are updated only by special-purpose maintenance tools that synchronize with Cheetah explicitly.

Knowledge-Based Packet Merging: This exploits knowledge of per-request state transitions to avoid sending distinct TCP control packets that can easily be merged (a.k.a., piggybacked) with quickly forthcoming data packets. Two examples of this are setting the FIN bit on the last data packet and explicitly delaying the ACK on clients' HTTP request packets. This reduces the number of distinct packets, making both the network and the end-points more efficient. It is particularly valuable for small document sizes, where the reduction represents a substantial fraction (e.g., 20–33%) of the total number of packets.

Precomputed Checksums: Cheetah precomputes the per-packet Internet checksums for the data in servable files and stores them in those files. These checksums are then provided with the data to the TCP library routine that constructs packets. This extension reduces the number of in-memory manipulations for transmitted data to zero (i.e., zero-touch disk to network transmission) when combined with the merged file cache/retransmission pool extension. The checksums are added to the servable files by the same maintenance tools discussed above for the merged file cache and retransmission pool.

As shown in Figure 4, Cheetah significantly outperforms the servers that use traditional interfaces. By exploiting application-level networking, Cheetah delivers 4 times the throughput for small documents (1 KB and smaller) and almost twice the throughput for large documents. Further, Cheetah's performance is limited by the network hardware in all cases. Specifically, small document performance is limited by the number of minimally-sized packets that can be pushed thru the system's network interfaces: measured at about 64K packets per second on our server system, or 8000 HTTP requests at 8 packets each. Large document performance is limited by the available network bandwidth (3 100Mbit/s Ethernets). In fact, while the Baseline implementation is limited to only 16.5 MB/s with 100% CPU utilization, Cheetah delivers over 29.3 MB/s with

xok-enabled Enhancement	100-byte	100KB
Direct access to cache/net	100 %	20 %
Packet merging	50 %	1 %
Merge cache/retransmit	2 %	40 %
Checksum precompute	1 %	10 %
Header precompute	20 %	<1 %
Total improvement	271 % (3.7X)	86% (1.9X)

Table 2: Estimated break-down of performance difference between Cheetah and Baseline HTTP server implementations (for 100 byte files and 100KB files). The total improvement represents the product of the improvements, since this breakdown assumes the improvements are cumulative.

the CPU idle over 30% of the time.

Cheetah Performance Breakdown. Table 2 provides a rough breakdown, based on additional system statistics and detailed profiling, of Cheetah's performance advantage over the Baseline implementation. The table shows the breakdown for two ends of the web document size spectrum (larger sizes are similar to 100KB in behavior). The medium-sized performance is explained as mixtures of these two end-points.

For the small request sizes, a 50% increase in throughput comes from the 33% reduction in packets transmitted/received (due to the knowledge-based packet merging enhancement). Profiling reveals that the Baseline server spends over half of its time in the various routines involved with implementing the general-purpose file descriptor and socket interfaces. Unfortunately for optimization purposes, the time is spread widely among 108 functions involved with implementing these very general abstractions; also, recall that the xok/ExOS implementations of these abstractions are already well-tuned relative to OpenBSD.

For the large request sizes (e.g., 100KB), the Baseline server spends about 30% of its time in bcopy and 8% in the checksum routine (as compared to 4% and 3% for Cheetah). Note that these percentages can be usefully compared by recalling that Baseline uses 100% of the CPU and Cheetah uses 70% of the CPU (for this document size); the only caveat is that Cheetah wastes much more time context switching back and forth with the idle task (which accounts for some of the 70% utilization). The remainder of the performance difference is again due to the layers of code involved with the POSIX file and socket interfaces.

Other Applications and Workloads. Although it is highly specialized, Cheetah shares system resources (e.g., CPU and memory) like any other application on a time-sharing system. As would be expected, we find that Cheetah has no significant effect on the performance of other applications when it is idle. Given a specific HTTP workload, other applications run faster when sharing a system with Cheetah than when sharing with the Baseline server, because Cheetah uses less CPU time (more efficient operation) and less main memory (no distinct retransmission buffers). With respect to more substantial web servers, this increases the CPU power available for useful services, such as CGI utilities and database searches.

We have also evaluated Cheetah with a more representative HTTP benchmark, called SURGE [5]. SURGE emulates the behavior, along many axes, of a configurable number of users concurrently browsing the web site under test. The results indicate that Cheetah's average response time is 87% lower (with 90% lower variance) than that of the Baseline server on xok/ExOS, whose average response time is, in turn, 70% lower than the same server on OpenBSD. For example, with 50 simulated users, the average response times for the two servers on xok/ExOS are 7 ms and 54 ms, respectively. With 100 users, they are 38 ms and 286 ms.

5.4 Discussion

Clearly, any performance enhancement that an application writer can make with xok/ExOS's application-level networking, a kernel programmer could make inside a conventional OS by replicating or complicating the default networking stack. Therefore, Cheetah's performance could certainly be achieved inside a conventional system, given OS source code, sufficient development and debugging time, and a willingness to live without protection boundaries. The power of application-level networking lies in its ability to let application writers manipulate networking features (e.g., those described here and the next, as yet unknown, set) that have traditionally been hidden away in OS kernels.

It is also interesting to consider which aspects of Cheetah's performance could be realized more incrementally in existing systems. For example, some of the extensions (e.g., delaying the first ACK) could simply be incorporated into a system's default TCP implementation. However, although it improves performance for Cheetah's behavior, delaying the first ACK results in undesirable retransmissions and window-shrinkage for other applications. The various packet combining enhancements could be selectively supported with new *ioctl* and *send* flags.

The general overhead of the socket and file system interfaces is unlikely to be reduced substantially – these aspects of *xok/ExOS* are already significantly optimized, as evidenced by their performance relative to OpenBSD. One possibility would be to introduce a variety of less general interfaces to avoid the overhead of handling numerous states and options. As discussed in Section 2, many copy-free interfaces could improve large file performance, though the buffers must be kept read-only for long periods of time if they are to double as retransmission buffers. Given appropriately malleable TCP and file system implementations, Cheetah's merging of file cache/retransmission pool, merging of the FIN packet and checksum precomputation could all be provided behind interfaces such as IO-lite [38] or Windows NT's "sendfile."

6 webswamp: a fast HTTP client

Servers are not always just passive accepters of connections; some also initiate connections (e.g., a proxy HTTP server fetching a missing page). This section describes a specific instance of exploiting extensibility for a connection-initiating service: a web server benchmarking tool, called webswamp. Several of webswamp's specializations apply generally to such services and to clients as well.

6.1 Specialization of the "Client-side"

Early in the development of Cheetah, it became clear that existing web benchmarks were not capable of overloading it without excessive equipment—either very powerful or very numerous client machines. Therefore, we used xok/ExOS's application-level networking to construct webswamp, a low-level core upon which web benchmarks with different offered workloads can be easily built. A benchmark is developed by constructing a high-level driver that the webswamp core calls to get the next web page and inter-request think time. We have built two high-level drivers, one that repeatedly requests the same document with zero think times and one that uses the Surge [5] algorithms to produce a more representative workload. The webswamp core emulates a number of concurrent clients, each of which operates in a closed loop of "thinking" and then requesting another document. The low-level core initiates connection establishment when think times expire, handles incoming packets and any timeouts, generates/sends HTTP requests when connections become open, acknowledges/counts data received, closes down connections that receive FIN bits, and calls up to the high-level driver to get the next task when a connection is closed.

In addition to avoiding the overheads associated with layers of socket and file descriptor interface code, webswamp benefits from three main specializations:

Count-only Data Reception. Webswamp avoids the need to copy received data that will be discarded. Like most web benchmarks, webswamp discards received data after counting it. Because it can directly count the valid data in each received TCP packet, webswamp does not need to copy the data into its own buffers in order to get a count. Thus, only the copying involved with *xok*'s kernel-emulated application-level networking interface prevents zero-copy data reception.

Skipping Checksum Verification. Webswamp can be configured to not verify the integrity of incoming packets, removing the overhead of checksum computation. While unacceptable for general activity and not appropriate when using webswamp to test correct operation under heavy loads, this does increase the potential offered load during benchmarks. In our current experimental testbed, this enhancement does not change end-to-end performance (it only increases client machine idle time), because the previously discussed enhancements are sufficient to make the network itself the bottleneck.

Knowledge-Based Packet Merging. As Cheetah does, webswamp can exploit knowledge of its activity to avoid sending distinct control packets that can be merged with other soon-to-be-sent packets. Specifically, the ACK on the server's SYN packet can be discarded because the client is going to immediately send a data packet (the HTTP request) on which the ACK can be piggybacked. Similarly, the ACK on the server's last data packet and FIN bit can be discarded, because the client is going to immediately close the connection and send a FIN packet on which the acknowledgements can be piggybacked. For small document sizes, these two packet merging enhancements reduce the total number of packets by 25%, on top of Cheetah's reduction in packet count. (Note: because this enhancement changes the client load placed on the server, it is not enabled for any of the experiments in the other sections.)

Like Cheetah's packet mergings, webswamp's improve end-to-end performance within the confines of a correct TCP implementation, but result in undesirable TCP behavior when used in other applications. An additional client-side packet merging, setting the FIN bit on the HTTP request data packet, could be very beneficial to web server performance under HTTP/1.0. While this would not reduce the number of packets more than the mergings mentioned above, it would transfer the task of supporting TIME_WAIT connections to the more numerous clients. This was not part of our experiments below.

6.2 HTTP Client Performance

To place webswamp's performance in context, we developed a socket-based version of the same core functionality. It supports the same interface with the high-level driver and uses non-blocking socket calls to connect to a server, send HTTP requests, and receive web pages.

Figure 6 shows the HTTP request throughput as a function of request size achieved by four different benchmark implementations: the socket-based client running on OpenBSD and on xok/ExOS, and webswamp on xok/ExOS without and with the packet-merging enhancement.

Webswamp supports much higher throughputs than the socket-based implementations (a factor of 5 higher for small HTTP documents and 15% for 10KB documents), even with xok/ExOS's superior socket performance (50–75% faster than OpenBSD for small documents and 22% for 10KB documents). The packets exchanged by client and server are exactly the same for all three implementations. The throughputs converge as the file size increases, because the single

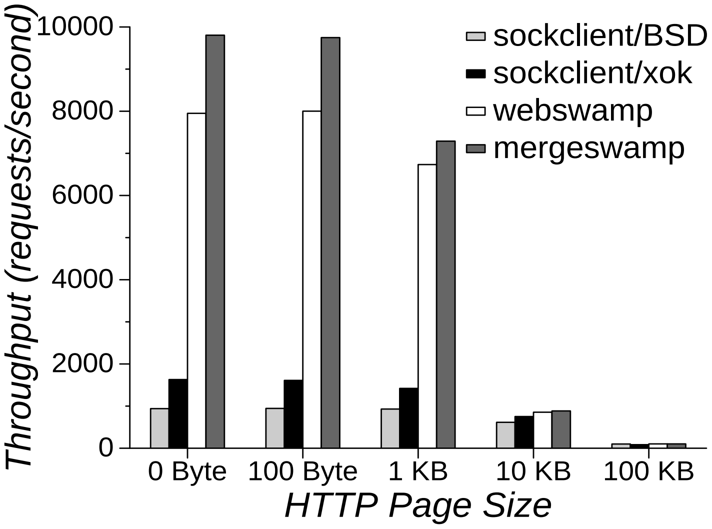


Figure 6: HTTP document throughput as a function of the document size for different client-side implementations benchmarking Cheetah. "Sockclient/BSD" and "sockclient/xok" represent the socket-based version running on OpenBSD and xok/ExOS, respectively. "Webswamp" and "mergeswamp" represent webswamp running on xok/ExOS without and with the packet merging extensions. The throughput values are obtained with 6 clients on a single machine repeatedly requesting the same web document with zero think time between the completion of one request and the initiation of the next.

(per-client) 100 Mbits/second link limits the bandwidth. Webswamp's performance advantage is almost entirely due to the avoidance of overheads related to the general POSIX socket interface.

Figure 6 also illustrates the performance benefit of knowledge-based packet merging on the client-side. For 1 KB and smaller HTTP documents, the number of packets transmitted by the client is reduced by 40% (from 5 to 3, representing a 25% reduction in the total number of packets per request). This translates into a throughput increase of 25% for 0 KB documents, 21% for 100 byte documents, and 8% for 1 KB documents. For 0 KB and 100 byte documents, we see again that the network interface's packet throughput limits end-to-end performance. For 1 KB documents, bandwidth limitations reduce the overall improvement. For 10 KB documents, the total number of packets is reduced by 10% but throughput only increases by 4%. The difference for 100 KB files is negligible.

Experiments with *surgeswamp*, the webswamp-based version of Surge, indicate that it can usefully simulate a much larger number of clients than the default pthread-based implementation. Also, the times at which surgeswamp initiates connections to the server more closely match those requested by the high-level decisions, which means that the workload is more representative of reality.

7 Application-level TCP Forwarder

With an example inspired by [19], we illustrate that extensible systems can be exploited to simultaneously realize performance, simplicity and semantics not possible with conventional I/O abstractions. The example is an application-level protocol forwarder that can be used to redirect

TCP connections to other hosts. Such functionality is useful for several purposes, the clearest being transparent load balancing of client requests among a set of server machines.

For our experiments, we implemented two protocol forwarders that act as intermediaries for connections to specific TCP ports. Both use the same code for deciding where to forward connections and different code for doing so. The first forwarder uses non-blocking sockets to listen for and accept client connections. After accepting a client connection and choosing a back-end server, the forwarder opens a second non-blocking connection to the chosen server. It passes received data from each to the other and closes both when either side closes. These actions are performed in parallel for multiple client connections. This socket-based implementation is expensive in that data passes through the protocol code and is copied about multiple times. More importantly, it violates the end-to-end semantics of the TCP protocol. For example, the client receives acknowledgements from the intermediary (rather than the server), independently of whether a server ever actually receives the corresponding information. This socket-based forwarder also interferes with TCP's algorithms for end-to-end flow control and congestion control, potentially reducing overall performance.

The second forwarder implementation exploits extensibility to provide both correct semantics and improved performance. This forwarder simply waits for a relevant packet to arrive and determines if it is for an open connection. If it is, the forwarder modifies the destination information (ethernet addresses, IP addresses and TCP port numbers), patches the IP and TCP checksums to reflect the changes, and transmits the packet out the appropriate network interface. If the packet is not for an existing connection and contains a SYN, the forwarder chooses a back-end server, initializes demultiplexing structures for the client and chosen server, and then forwards the packet as described above. With this implementation, the forwarder avoids processing of most packets, acting mainly as a high-level router. More importantly, it provides correct end-to-end TCP semantics, and its presence is invisible to the client.

Figure 7 shows the performance of these two forwarders acting as intermediaries between webswamp on one fast ethernet and Cheetah on a second fast ethernet. Significant performance improvements are realized in the specialized direct-forwarding implementation by eliminating extraneous packet processing/generation (for small requests) and copying (for large requests), when compared to the socket-based forwarder. This translates into a factor of 3 increase in throughput for small document sizes and 40–50% for larger document sizes. As with Cheetah and webswamp, the performance of the specialized forwarder is limited by the network rather than software overheads.

8 Discussion

The construction and revision of the xok/ExOS networking support came with several lessons and controversial design decisions. This section discusses a number of these:

Event Notification. All event notification in xok/ExOS's network services is based on polling and WK predicates. This form of event notification is a powerful foundation for decentralization, allowing processes to wait for and be awakened on events without having to tell any other processes about them. Likewise, processes can awaken processes by simply updating memory normally, and they do not have to know who, if anyone, is waiting for the update. Still, our experiences have not been all positive. In addition to the inherently higher event discovery latency, we have observed two forms of scalability problems with WK predicates. First, the size of a WK predicate grows linearly with the number of events of interest to an application, which effects the storage space, the time to install the predicate, and the time for the scheduler to

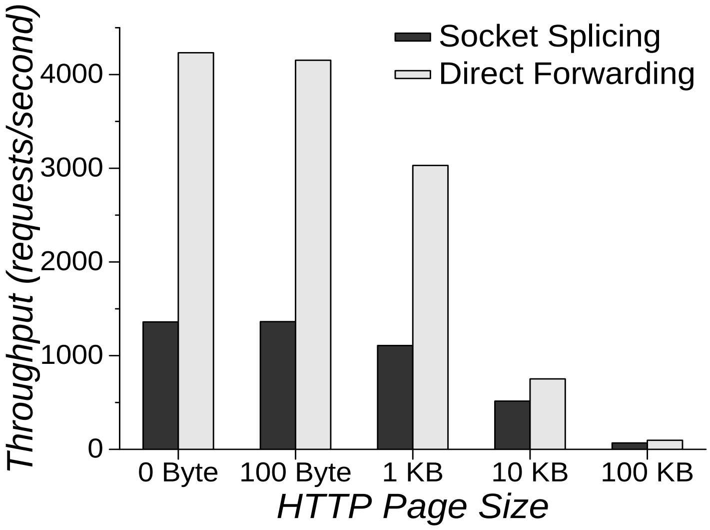


Figure 7: HTTP document throughput as a function of the document size for application-level TCP forwarding (from a client on one machine through the forwarder to an HTTP server on a third machine). "Socket Splicing" moves data between a pair of sockets communicating with the client and server, respectively. "Direct Forwarding" simply patches and forwards packets directly to the desired recipient, reducing redundant protocol processing and data manipulation. In addition to the performance improvements, "Direct Forwarding" provides correct end-to-end TCP semantics while "Socket Splicing" does not. The throughput values are obtained with 6 clients on a single machine repeatedly requesting the same web document with zero think time between the completion of one request and the initiation of the next.

check it. Second, the number of WK predicates checked by the scheduler grows linearly with the number of processes in the system, since most processes in a system are waiting for events at any given point in time. For systems with many processes and processes interacting with many services (e.g., consider a select on 1000 connections), these are significant problems. For networking, more explicit notification could be incorporated without hurting decentralization, since the main events of interest come from the kernel (packet arrived, packet transmitted, time advanced). Done efficiently, more explicit notification is probably the correct approach for application-level networking.

DPF. To demultiplex incoming packets, some form of packet filter engine is required—xok/ExOS uses DPF [16], which uses dynamic code generation to improve filter checking speeds. While DPF offers improved performance and supports arbitrary filters (i.e., filters that claim packets based on arbitrary fields in packets), DPF's complexity has had a negative impact on the evolution of xok/ExOS. For example, moving from the MIPS platform to the x86 platform involved significant work on DPF. Similarly, when bugs and shortcomings are encountered in the x86 DPF, the latency of fixes is quite long because DPF's complexity limits the set of people who can manipulate its internals. Finally, the flexibility offered by allowing arbitrary filters is of questionable value in the real world and creates an unsolved dilemma: what to do about overlapping filters. For example, if filter A checks only byte #1 and filter B checks only byte #2, there is no clear way to know whether both should be allowed and which should win ties. An explicit goal of the exokernel project was to avoid having the kernel understand any specific network protocols. However, since all communication on a given network must use a consistent demultiplexing scheme, exploiting this information to eliminate difficult problems makes too much sense.

Process Scheduling. xok's process scheduling mechanism does not explicitly consider interactive responsiveness. With application-level networking, this can have a negative impact on packet processing latencies. While xok's scheduler will give the CPU to an awakened process whose turn it is to run, it will not preempt another process's turn. While this is more fair for processes that care when they run, it does create performance problems for some forms of ping-pong communication. Appropriate CPU scheduling support for application-level networking remains an open area for research.

Unchecked Transmission. xok does not examine the contents of outgoing packets, allowing applications to transmit any data desired onto the network. This controversial choice increases flexibility, increases performance, and simplifies the implementation. However, it is clearly a problem if one is trusting packets on the wire, and one inappropriately believes that none of the other machines on the network allow some applications to transmit arbitrary packets (as most do). Unfortunately, many environments still do make these faulty assumptions. Therefore, most modern operating systems restrict the ability to send arbitrary packets to processes with "root" privileges. The two known approaches to providing this same level of protection with application-level networking are NI-attached headers and reverse packet filters. The former somewhat reduces flexibility and only ensures that the added headers are correct. The latter involves checking the appropriate header values of each packet to be sent and then ensuring that the headers are not changed, which requires blocking writes or copying the headers. Still, several system vendors have indicated a desire to see the protection in place, simply to increase the degree of difficulty for bad people.

Transmit Queue Fairness. Like most operating systems, xok's current implementation enforces no fairness with respect to the transmit queue. For guaranteed rate or real-time communication, in particular, this is an important consideration. Fixing this problem requires explicitly treating the different applications as distinct sources of packets, and only letting them add to a transmit queue when it is "their turn" according to the chosen policy. Nemesis [7] and U-net [49] have both demonstrated that this type of fairness can be accomplished by allowing the NI explicitly schedule packets from separate per-application or per-connection transmission queues.

Inter-Endpoint Resource Sharing. As discussed in Section 3, we have found inter-endpoint resource sharing to be critical to performance and scalability. For example, allowing a single pre-posted packet buffer to be used for any of a large set of open connections allows application-level networking software to scale memory usage more like kernel implementations. Unfortunately, the emerging Virtual Interface Architecture standard [1] for application-level networking is inappropriately preventing such sharing. Unchanged, this may limit the VI Architecture's applicability to the small set of parallel computing applications with which it started.

Auxiliary Information about Packets. Although xok does not do so, we believe that there is value in having additional values added to received packets at the point that they are copied into the packet rings. Two specific additions of value are an identification of which card received the packet and the time of that reception. The card identification can be valuable for authenticating packet sources. The packet reception time would allow round-trip time measurements to not be obfuscated by process scheduling delays.

9 Conclusions

This paper describes the architecture, implementation, performance, and flexibility of xok/ExOS's application-level networking. This system has demonstrated that network services such as TCP/IP, UDP/IP, POSIX sockets, ARP, DNS, and tcpdump can be implemented as independent application-level libraries. In addition, applications can safely employ highly specialized networking software to achieve substantial performance and semantic improvements.

We hope that our successes and failures can help in the design of emerging standards for the architecture and interfaces for NIC-multiplexed network interfaces. Poorly-designed protection mechanisms can make safe application-level networking complex, slow and/or impossible. Well-designed protection mechanisms can provide application-level library writers with the flexibility and functionality required for safely providing core network services, while also enjoying the massive performance and functionality advantages (illustrated in [47, 14, 49] and Section 5.3) possible with application-level networking.

Acknowledgements

We thank the past and present members of MIT's Parallel and Distributed Operating Systems group, who helped to develop the ideas and xok/ExOS system described in this paper. Special thanks to Josh Smith for helping us with a variety of the benchmark construction and data collection tasks. We also thank Paul Barford for providing us with a pre-release copy of Surge.

This research was supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288.

References

[1] The virtual interface (vi) architecture. http://www.viarch.org/, 1998.

[2] T. Anderson. The case for application-specific operating systems. In *Third Workshop on Workshop Operating Systems*, pages 92–94, 1992.

[3] G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Workshop on Internet Server Performance*, June 1998.

[4] G. Banga and J. Mogul. Scalable kernel performance for internet servers under realistic loads. In *USENIX Technical Conference*, June 1998.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Conference*, June 1998.

[6] 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.

[7] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *IEEE Conference on Local Computer Networks*, 1997.

[8] R. Braden. T/tcp – tcp extensions for transactions functional specification. RFC 1644, USC/Information Sciences Institute, July 1994.

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

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

[11] Digital semiconductor 21140A PCI fast ethernet LAN controller hardware reference manual. Digital Equipment Corporation Publication Number EC-QN7NE-TE, November 1996.

[12] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *OSDI*, pages 261–276, oct 1996.

[13] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM SOSP*, pages 189–202, December 1993.

[14] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on low-cost gb/s lan. In *ACM SIGCOMM 1994*, pages 14–23, September 1994.

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

[16] D. Engler and M.F. Kaashoek. Dpf: fast, flexible message demultiplexing using dynamic code generation. In *ACM SIGCOMM 1996*, pages 53–59, August 1996.

[17] T. Faber, J. Touch, and W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *INFOCOM*, pages 1573–1583, 1999.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Proposed Standard RFC 2068, HTTP Working Group of IETF, January 1997.

[19] M. Fiuczynski and B. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX Technical Conference*, pages 55–64, January 1996.

[20] 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.

[21] J. Heidemann. Performance interactions between p-http and tcp implementations. ACM Computer Communication Review, April 1997.

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

[23] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[24] K. Johnson, M.F. Kaashoek, and D. Wallach. Crl: High-performance all-software distributed shared memory. In *15th ACM SOSP*, pages 213–228, December 1995.

[25] M.F. Kaashoek, D. Engler, G. Ganger, and et al. Application performance and flexibility on exokernel systems. In *16th ACM SOSP*, pages 52–65, October 1997.

[26] M.F. Kaashoek, D. Engler, D. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, pages 141–148, September 1996.

[27] O. Krieger, M. Stumm, and R. Unrau. The alloc stream facility: A redesign of application-level stream i/o. *IEEE Computer*, pages 75–82, March 1994.

[28] B. Lampson and R. Sproull. An open operating system for a single-user machine. In *7th ACM SOSP*, pages 98–105, December 1979.

[29] A. Luotonen, H. Frystyk, and T. Berners-Lee. CERN HTTPd. http://www.w3.org/pub/WWW/Daemon/.

[30] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM SOSP*, pages 244–255, December 1993.

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

[32] M.K. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, 1996.

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

[34] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *11th ACM SOSP*, pages 39–51, November 1987.

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

[36] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Trans. on Computer Systems*, 10(2):110–143, May 1992.

[37] V. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: A unified i/o buffering and caching system. Technical Report CS Technical Report TR97-294, Rice University, 1997.

[38] V. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: A unified i/o buffering and caching system. In *OSDI*, pages 15–28, feb 1999.

[39] 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.

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

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

[42] V. Jacobson S. McCanne. The bsd packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX*, pages 259–269, January 1993.

[43] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *OSDI*, pages 213–228, oct 1996.

[44] SMC EtherPower 10/100 fast ethernet PCI network card user guide. Standard Microsystems Corporation Publication Number 79-000668-001, 1996.

[45] User guide for SMC's TigerSwitch 100. Standard Microsystems Corporation Publication Number 900.168, November 1996.

[46] S. Spero. Analysis of http performance problems. http://sunsite.unc.edu/mdma-release/http-prob.html.

[47] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *ACM SIGCOMM 1993*, pages 64–73, September 1993.

[48] Virtual interface architecture specification. Compaq, Intel, Microsoft, December 1997.

[49] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *15th ACM SOSP*, pages 40–53, 1995.