Migrating Sockets—End System Support for Networking with Quality of Service Guarantees*

David K.Y. Yau
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
yau@cs.purdue.edu

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
lam@cs.utexas.edu

August 14, 1998

Abstract

We present an end system architecture designed to support networking with QoS guarantees. The protocol processing component of the architecture, called Migrating Sockets, has been designed with minimal hidden scheduling which enables accurate determination of the rate requirement of a user application. The end system provides QoS guarantees using (i) an adaptive rate-controlled scheduler, (ii) rate-based flow control on the send side for access to reserved-rate network connections, and (iii) a constant overhead active demultiplexing mechanism on the receive side which can be transparently enabled in wide-area TCP/IP internetworking (although it is not restricted to TCP/IP). To achieve efficiency, Migrating Sockets lets user applications manage network endpoints with minimal system intervention, provides user level protocols read-only access to routing information, and integrates kernel level support we previously built for efficient data movement. Migrating Sockets is backward compatible with Unix semantics and Berkeley sockets. It has been used to implement Internet protocols such as TCP, UDP and IP (including IP multicast), and run existing applications such as *vic*. Migrating Sockets has been implemented in Solaris 2.5.1. We discuss our implementation experience, and present performance results of our system running on Sun Sparc and Ultra workstations, as well as Pentium-II desktops.

*Research supported in part by National Science Foundation under grant no. NCR-9506048, an equipment grant from AT&T Foundation, and an IBM graduate fellowship. Part of this work was done while David Yau was with the University of Texas at Austin. An earlier version of this paper was presented at the IEEE ICNP '97 conference.

1 Introduction

It is increasingly important for end systems to provide support for networking with quality of service (QoS) guarantees. This trend is in part due to the emergence of continuous media (such as video and audio) applications having real-time constraints. Such work on end system support complements recent research on integrated services networks. QoS guarantees provided by network level packet scheduling and admission control [2, 7, 29] can thus be extended to the ultimate endpoints of an end-to-end communication, namely applications running in user space of general purpose operating systems.

End system support for networking with QoS guarantees is a challenging problem. To meet various timing constraints, user processes must be given guaranteed access to diverse system resources, including time-shared resources such as CPU and network interface, and space-shared resources such as memory buffers. Moreover, the run time environment for protocol processing, which provides such services as timer management, buffer management and demultiplexing table lookup, should be designed to support predictable performance.

Besides QoS guarantees, recent proliferation of heterogeneous networking technologies and user application requirements [11, 12, 18, 29] will make customized development and flexible deployment of network protocols highly desirable. Protocol implementation at user level can help achieve these goals. With fault containment in user processes and the availability of sophisticated tools for developing user level code, the cost of protocol development and experimentation will go down, and the lead time to deployment of protocols in a production environment will be reduced [24]. Moreover, without the need to configure and load protocols into kernel space, user applications can be given access to a wider choice of protocol stacks and select ones that are most appropriate to their needs.

1.1 Our contributions

We discuss our experience implementing Migrating Sockets as a framework for user level protocols that can run with guaranteed progress. Our system integrates many ideas for providing QoS guarantees and achieving implementation efficiency. Some of these ideas are novel while others have been discussed in prior work [16, 24, 27, 28]. We believe that integrating all these ideas into a single working system is a contribution in itself.

Of the many techniques implemented in our system, we consider the following ones to be novel. First, Migrating Sockets has been designed to minimize *hidden scheduling* in protocol processing.[1] This allows protocol threads to run with well-defined rate requirements which are met in our system by adaptive rate-controlled (ARC) scheduling. Second, "exclusive packet receiver" information exported by Migrating Sockets enables a constant overhead packet demultiplexing mechanism called *active demultiplexing*. By eliminating table search, active demultiplexing is highly efficient and has predictable performance. Third, we introduce *delayed caching* and *recall-on-access* for socket migration which reduce connection management overhead for the widely used concurrent server model in client/server programming. Fourth, we introduce the use of a well-known shared memory region to enable efficient sharing of routing information between network and higher layer protocols in a user level implementation model. Fifth, we have implemented a *thread folding* mechanism, whereby receive side protocol processing can be performed by an application thread reading directly from the network. This saves one context switch and simplifies determination of the application's rate requirement.

1.2 Related work

User level protocol implementation was first proposed and investigated by Thekkath et al [24] who demonstrated that it can be as efficient as a kernel-resident implementation. The idea of using a server process for connection management and allowing client processes to cache socket states for efficient network access was proposed by Maeda and Bershad [16], who also investigated issues in maintaining backward compatibility

[1] Hidden scheduling occurs when protocol processing is done in the context of interrupt handling or background threads of control that do not belong to a user process.

with Unix semantics. In designing Migrating Sockets, we learned from their experience. Many other systems also implemented protocols at the user level [3, 5, 25]. However, unlike our end system architecture, the issue of system support for quality of service guarantees was not a concern in any of these papers.

We next review several systems whose design objectives are similar to ours, namely, QoS support and efficient implementation. These systems (including ours) were designed for OS platforms with different capabilities. As a result, different design decisions were made leading to a variety of solutions for satisfying the objectives.

User level protocols are used in [14, 15] to provide QoS support for multimedia applications. The *processor capacity reserve* abstraction in Real-Time Mach is used for resource reservation and scheduling. Real-time Mach provides scheduling algorithms such as rate-monotonic and earliest-deadline-first. To protect well-behaving applications from mis-behaving ones, processor capacity reserves are periodically replenished and threads that have overrun their reserves are given a *depressed* priority, and henceforth receive different scheduling. We took a different approach in our design by employing rate-based schedulers with a provable firewall property for protection between threads. (Neither rate-monotonic nor earliest-deadline-first has the firewall property.) Furthermore, bandwidth scheduling and packet demultiplexing are major concerns in our end system architecture. These issues are not addressed in [14, 15].

A real-time upcall (RTU) mechanism is used in [8] to implement user level protocols with QoS guarantees. The system is built on top of NetBSD, whose kernel is not multithreaded. Designing a system without use of threads led to an event-driven approach, in which RTUs are handlers for protocol events. Their resource reservation model is based on the number of packets processed per time interval. RTUs can respond to protocol events in a timely manner. By delaying preemption until packet boundaries, concurrency in protocol processing is achieved without using locks which contributes to performance gains. Our system is very different in at least two respects. First, our resource reservation model is a CPU rate without any reference to packet processing. Second, we tackled the thread scheduling problem; in particular, priority inheritance is implemented to solve priority inversion that can arise from lock contention.

Aqua [13] proposes integrated CPU and network IO scheduling in Solaris. It uses in-kernel Solaris protocol stacks for protocol processing and an "extended rate-montonic" algorithm for CPU scheduling (both design choices are different from those in Migrating Sockets). For network IO in Aqua, packets are divided into a real-time class and a best-effort class. When real-time packets are received from the network, they are eagerly processed in interrupt context, as usual. When best-effort packets are received, however, interrupt processing is curtailed, and any unfinished work is later handled by a background streams service routine at a lower priority. In this way, a two-level differential service is provided. Rather than a differential service, our system uniformly minimizes hidden scheduling (due to interrupt processing as well as background facilities such as streams service routines) in order to provide guaranteed performance at the level of thread scheduling.

An idea similar to active demultiplexing can be found in the buffer queue index (BQI) approach of Autonet-2 [4], a high performance switching system built at DEC SRC. Unlike Autonet-2, active demultiplexing is strictly a software approach and does not require special support from network switches or interfaces. Also, we show that active demultiplexing can be efficiently implemented in wide area internetworking, such as using TCP/IP.

1.3 Organization of this paper

In section 2, we give an overview of our end system architecture for networking with QoS guarantees. We show how Migrating Sockets works together with other system components to provide QoS guarantees in an end-to-end path of network communication. Section 3 introduces Migrating Sockets as a user level protocol implementation framework. The framework is flexible, efficient and backward compatible with Unix semantics and Berkeley sockets. Section 4 describes several novel aspects of Migrating Sockets. ARC scheduling of protocol threads is presented in section 5. Active demultiplexing is described in section 6. We present micro and macro-benchmark results of our current system in section 7 and conclude in section 8.

2 Architectural Overview

Our end system architecture for networking with QoS guarantees has the following major components: (1) ARC scheduling for time-shared resources in an end system, such as CPU and network interface, (2) a Migrating Sockets framework for user level protocols that minimizes hidden scheduling in protocol processing, and (3) a constant overhead packet demultiplexing mechanism suitable for wide area internetworking. Figure 1 illustrates the architecture. A packet path from the sender system on the left to the receiver system on the right is shown.

Migrating Sockets (see section 3) takes its name because the state and management right of a network endpoint can move between a network server and client processes. With Migrating Sockets, performance critical protocol services such as send and receive are accessed as a user level library linked with applications. Send side protocol code is accessed in usual application threads of control. In addition, user processes have protocol threads for network receive and timer processing. From a QoS perspective, Migrating Sockets has the advantage of minimizing hidden scheduling. For example, the role of network interrupt handlers in our system is only to deliver packets to a set of destination processes. The bulk of receive side processing is done in the context of a user thread of control.

With hidden scheduling minimized, user applications can more easily determine and negotiate an appropriate *rate of progress* with an end system, such that their real-time constraints can be met. In our architecture, progress requirements are specified with two parameters: a reserved *rate* (between 0 and 1) and a time interval known as *period* (in μs). Based on the progress requirements of all threads in the system, an ARC CPU scheduler can perform admission control and provide *conditional* progress guarantees to threads. One form of progress guarantees provided by the ARC scheduler [28, 26] is as follows: a "punctual" thread with rate r and period p is guaranteed at least krp CPU time over time interval kp, for $k = 1, 2, \ldots$.

On the send side, ARC scheduling enables applications to respond to media events and generate network packets "in time." These packets then enter a network connection, which may have a reserved rate negotiated with an integrated services network. If the network connection is shared by multiple processes, it is possible for a sudden burst of packets by one process to block out access to the network connection for an extended period of time [27], thereby jeopardizing the bandwidth requirements of other processes. This problem is especially pronounced if the connection has a moderate or low reserved rate. To solve the problem, an end system should provide rate-based flow control to reserved-rate network connections. In our proposal [27], flow control is enforced by a *lightweight kernel thread*. The approach is quite flexible in that different flow control policies can be provided by different loadable kernel modules.

On the receive side, packet arrivals to a network interface are processed by the interrupt handler of the interface. Kernel level code must then demultiplex the packets to their destination processes. Traditionally, such demultiplexing is performed by packet filters [6, 17] (also known as packet *classifiers* in [1]). Our system makes use of packet filters, but, in addition, can exploit "exclusive packet receiver" information exported by Migrating Sockets to perform *active* packet demultiplexing. A network endpoint that is an exclusive packet receiver has the property that packets destined for it should not be delivered to any other endpoint in the system.

In active demultiplexing, an exclusive packet receiver *advertises* to a peer sender an OS handle for packet delivery. On learning the advertisement, the sender *encloses* this OS handle in packets it sends to the receiver. The kernel demultiplexing code in the receive system can then make use of the handle to deliver packets directly to the receiver, without table searching. For safety reasons, the receive kernel checks that a handle is indeed associated with an exclusive packet receiver and ensures the "freshness" of the handle by using a *nonce* included with the handle. From a QoS perspective, active demultiplexing is desirable since it is a constant overhead mechanism, contributing to predictable performance.

In the receive end system, demultiplexed network packets may cause their receiver processes to be scheduled. An ARC CPU scheduler in the receive system enables such processes to respond to the packet arrivals "in time."

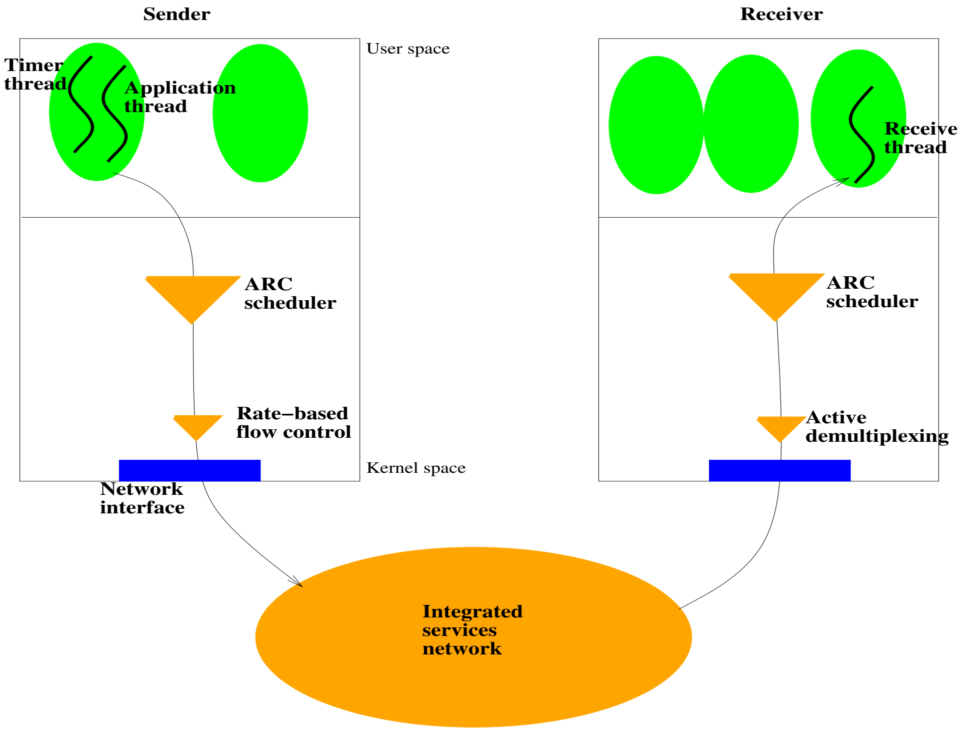

Figure 1: End system architecture for networking with QoS guarantees.

3 Migrating Sockets

In choosing an application programming interface (API) for our system, one of our goals is that the API should allow us to run existing and future Unix multimedia applications (such as the mbone suite of teleconferencing tools) with minimal modifications. We therefore decided to maintain backward compatibility with the widely used Berkeley socket interface. In concept, our Migrating Sockets framework draws upon previous experience in user level protocol implementation [16, 24]. In what follows, we give an overview of Migrating Sockets. Several novel ideas implemented in the framework are presented in section 4.

3.1 Berkeley sockets

Sockets are used by applications to access local endpoints of network connections. Function calls in the socket API can be classified into three major categories: (i) those for connection management, such as `socket()`, `bind()`, `listen()`, `accept()`, and `connect()`, (ii) those for sending and receiving data, such as `send()`, `sendto()`, `sendmsg()`, `recv()`, `recvfrom()`, `recvmsg()`, and (iii) those for managing endpoint characteristics, such as `ioctl()`, `setsockopt()` and `getsockopt()`.

We elaborate somewhat on connection management, since it is needed to understand caching and flushing of network endpoints in Migrating Sockets. An application opens a socket by using the `socket()` call, which returns a *socket descriptor*. For datagram (i.e. connectionless) communication, the application then uses `bind()` to associate a local address with the socket, which can then be used for sending and receiving data. For connection oriented communication, further operations after `bind()` are needed to establish network connections before communication can occur. This can be done by either a *passive* open or an *active* open. In a passive open, a server application uses the `listen()` call to declare its intention of receiving incoming connection requests. When a connection request arrives from a remote endpoint for the socket in listen state, a network connection will be established and a new socket will be created to access the newly established connection. The server application can then use the `accept()` call to get a socket descriptor for the new socket. In an active open, a client application uses the `connect()` call to request connection with a remote endpoint. When `connect()` returns, a network connection will have been established, accessible through the socket used in `connect()`.

Besides giving access to communication functions, socket descriptors are more generally file descriptors in Unix. Hence, sockets should support Unix semantics for file descriptors. First, it should be possible to have shared access to sockets by multiple processes, such as after the `fork()` system call, or after socket descriptors have been passed from one process to another. Second, the `select()` system call, which waits for input from any of a set of file descriptors, should work for socket descriptors as well.

3.2 Endpoint management

Figure 2 gives an overview of the protocol implementation model of Migrating Sockets. The model uses a network server process for "boundary" protocol operations like opening and closing network connections. User applications are client processes in the model. They register themselves with the network server. When a client process opens a socket, it creates a local data structure for the socket and contacts the network server with an RPC call. The network server likewise creates a data structure for the socket, and performs most of the "real" work for socket creation. Apart from using the `socket()` call, client processes can *implicitly* gain access to socket descriptors, such as after a fork. In the case of a fork, the network server copies socket descriptors from the parent process to the child process and uses callback RPC to ask the child process to create data structures for the sockets inherited. In summary, a client process knows about all the sockets for which it holds a socket descriptor, whereas the network server knows about all the sockets in an end system.

Since the network server has global knowledge of all the network endpoints in a system, it is suitable for the tasks of connection management. For example, it will be able to enforce uniqueness of network connections requested by different client processes. However, it is expensive to go through the network server for every socket operation. This is especially true of operations on the "performance critical" path, namely sending and receiving network data. In Migrating Sockets, after a network connection has been established, the state and management right of a socket can be *cached* to the client process holding an exclusive socket descriptor for the socket. Caching involves transferring the current state of the socket (including any data buffered for send and receive, and any protocol state associated with the socket) from the network server to the client process. A client process can then send to and receive from the cached socket without going through the network server.

For connectionless protocols, such as UDP, a socket can be cached after the `bind()` system call, which fills in a local address and port number for the socket. For connection oriented protocols, such as TCP, a socket can be cached after the `accept()` system call, which fills in both local and remote addresses and port numbers for the socket.

A socket continues to be cached until a condition unfavorable to caching occurs, at which time the socket is *flushed* to the network server. This involves transferring the state and management right of the socket from the client process back to the network server. As an example, flushing is required before a fork, since cached access is incompatible with the semantics of sharing network endpoints. It is also required before a close. After flushing, the network endpoint may continue to exist within the network server. Therefore, flushing before a close takes care of the requirement of certain protocols (such as TCP) that the lifetime of a network endpoint may exceed the lifetime of the process having access to the endpoint. All operations on a flushed socket go through the network server, using an RPC interface.

Whereas shared access to a socket can prevent the socket from being cached, caching may be re-enabled by the `close()` operation. The situation occurs if, after a close by some process, another process becomes the only one holding a socket descriptor to the socket in question. When that happens, our system caches the socket to the latter process.

Besides the cache and flush operations, our system supports a third operation for socket migration known as *recall*. The operation is a callback RPC for the network server to ask a client process to transfer the

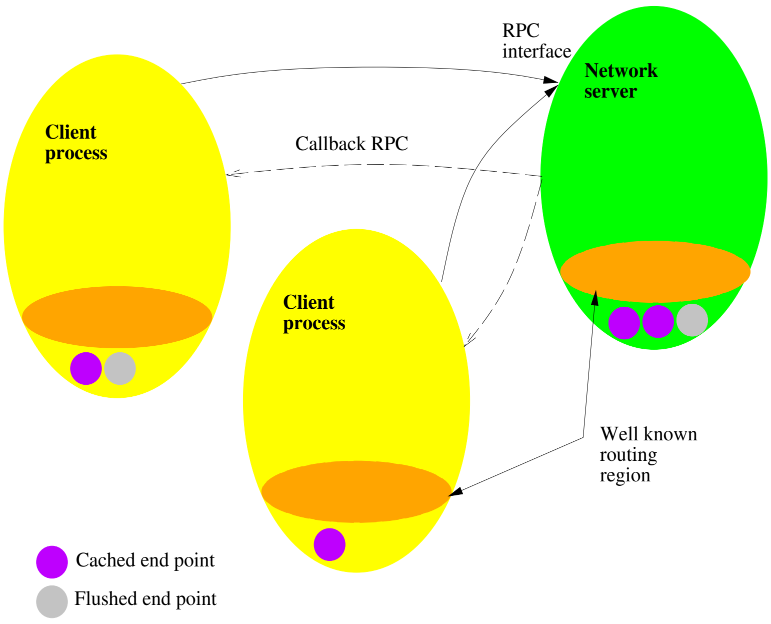

Figure 2: The protocol implementation model of Migrating Sockets.

state and management right of a cached socket back to the network server. It is needed, for example, in the optimization described in section 4.2.

3.3 Implementation considerations

Our implementation of Migrating Sockets leverages protocol code from 4.4 BSD. However, parts of the runtime support system have been rewritten. First, we replaced BSD mbuf buffer management by *message blocks* similar to those used in SVR4 streams. This is because mbuf has been found to treat small and large messages non-uniformly and hence exhibit undesirable performance idiosyncrasies [9]. Moreover, message blocks can very naturally handle both normal data buffers and *network buffers* (see section 4.4) supported in our system (using the `esballoc()` library call).

Second, we implemented a timer management interface for timer activities. Unlike 4.4 BSD, timer processing is driven by a timer thread of control.

Third, it is not feasible to protect critical code sections by raising interrupt level in a user level implementation. Instead, we make use of mutex locks and condition variables for mutual exclusion and condition synchronization. The current locking granularity is quite coarse. To illustrate, it is often convenient to think of protocol processing as consisting of an "upper" and a "bottom" half. The upper half is driven by protocol send activities, while the bottom half is driven by packets received from the network. Figure 3 shows the multi-threaded structure of a typical client process accessing cached sockets. A timer thread and application threads with send side protocol code run in the upper half. Protocol receive threads run in the bottom half. Notice that we use a thread of control for receiving from each cached endpoint.

In our locking model, a mutex lock `syslock` protects system data structures that are not modified by bottom-half threads. The purpose of the lock is then to synchronize access by multiple upper-half threads. For example, an upper-half thread opening a socket and another one doing a close may both be trying to modify a set of file descriptors associated with a process. Locking is required to serialize the two operations. Another mutex lock `intrlock` protects system data structures that may be modified by bottom-half threads. Its purpose is to synchronize access by bottom-half threads, as well as between the upper and bottom halves.

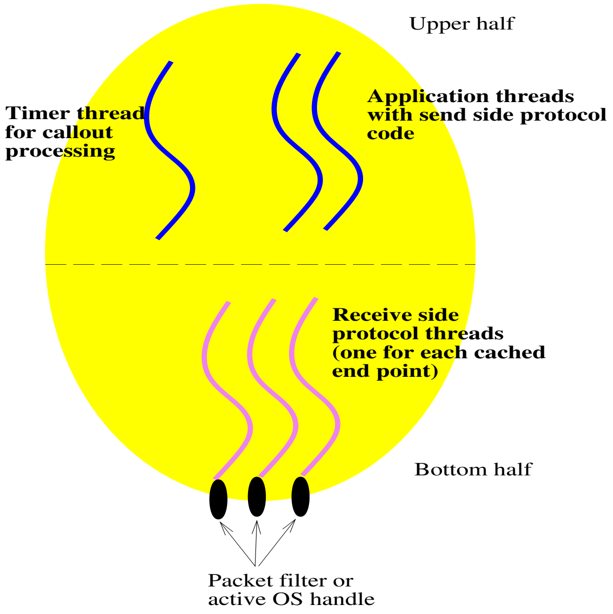

Figure 3: Multi-threaded process structure for applications accessing cached sockets.

For example, a bottom-half thread may be trying to append data to a receive socket buffer, while an upper-half thread may be trying to remove data from it.

Upper-half protocol threads normally acquire `syslock` before `intrlock`. Consistent use of this locking order avoids deadlocks between threads in acquiring the two locks. There are situations, however, in which the normal locking order is not followed. For these situations, the code fragment shown in Figure 4 is used to avoid deadlocks. Specifically, an upper-half thread is trying to read from a socket. The thread first acquires `syslock` and then `intrlock` before it checks the receive socket buffer. If it finds no data available for reading, the thread blocks on a condition `readable` protected by `intrlock` (`intrlock` will be automatically released when blocking occurs inside `cond_wait()`). Before the thread calls `cond_wait()`, however, it explicitly releases `syslock` to allow other upper-half threads access to data structures protected by `syslock`.

When the thread wakes up because data have arrived, it will have acquired `intrlock` while also needing to re-acquire `syslock`. In the figure, the call `mutex_trylock(syslock)` tries to acquire `syslock` but returns a failure condition of 0 if the lock cannot be acquired. Notice that if `mutex_trylock()` succeeded, the thread will have acquired both `syslock` and `intrlock`. If `mutex_trylock()` failed, however, the thread first releases `intrlock` and then tries again to acquire both locks in the normal order of `syslock` before `intrlock`, thus avoiding the possibility of deadlock with another thread.

4 Novel Aspects of Migrating Sockets

Our Migrating Sockets framework exports information on *exclusive packet receivers*, which has the following meaning: If a network endpoint is an exclusive packet receiver, then packets destined for it should not be delivered to any other endpoint in the system. Notice that in Migrating Sockets, network connections are established through the network server, which knows about all the existing network endpoints in the system. Hence, when the network server allows a socket to be cached, it knows whether the socket being cached is an exclusive packet receiver or not.

Algorithm LOCKING

```
mutex_lock(syslock);
...
mutex_lock(intrlock);
...
while (no data available for reading) {
        mutex_unlock(syslock);
        cond_wait(readable, intrlock);
        if (mutex_trylock(syslock) == 0) {
                mutex_unlock(intrlock);
                mutex_lock(syslock);
                mutex_lock(intrlock);
        }
...
}
```

Figure 4: Locking algorithm for deadlock avoidance.

Information on exclusive packet receivers can be used to reduce the search time for matching packets with packet filters, since a match with the filter of such a receiver means that further matching would be unnecessary. Moreover, as we will show in section 6, the information enables a constant overhead packet demultiplexing mechanism known as *active demultiplexing*.

In the following subsections, we elaborate upon several other novel aspects of our system, namely: (1) minimizing hidden scheduling in protocol processing, (2) caching optimization for the concurrent server programming model, (3) sharing of routing information between network and higher level protocols using a well-known shared memory region, and (4) a thread folding mechanism on the receive side for reducing context switch overhead and simplifying application progress rate determination. We also describe our kernel/user interface which provides user level protocol code with access to efficient kernel level support [27] through Unix file descriptors.

4.1 Minimizing hidden scheduling

Our experience [28] has been that it is difficult to provide QoS guarantees in certain protocol implementation frameworks. In streams [21], for example, network send and receive can take place in *service routines* run by "background" system threads of control. In BSD Unix, a single system timeout invocation has to handle outstanding timer activities of all the network endpoints in the system. The main problem of these background system services is that there is no easy way to determine suitable reserved rates of progress for the system services, such that the real time constraints of user applications can be met.

Aside from the use of background system services, traditional kernel level protocols perform entire receive side protocol processing in the context of interrupt handling. From a QoS perspective, it is similarly difficult to control the rate of progress of interrupt handling code (some researchers, such as [20], have considered disabling device interrupt for more predictable performance).

Migrating Sockets reduces the use of such hidden scheduling for cached sockets to a minimum. First, each user process has a dedicated timer thread that handles timer events only for network endpoints local to the process. Second, the role of the network receive interrupt handler in our system is minimal, i.e., only to demultiplex packets to their destination processes. Receive side protocol processing is done in the context of the receive thread associated with a cached endpoint.

```
Algorithm CSERVER

begin
        ...
        /* socket fd is used for accepting service requests */
        listen(fd, ...);
        while (1) {
                /* Request served through socket newfd */
                newfd = accept(fd, ...);
                if (fork()== 0) { /* child */
                        close(fd);
                        /* serve request */
                        ...
                        exit(0);
                }
                else /* parent */
                        close(newfd);
        }
...
end
```

Figure 5: Concurrent server using sockets.

4.2 Optimization for concurrent server model

In client/server programming, there are two principal programming models. They are the iterative server model and the concurrent server model (see, for example, [22]). In the latter model, the server's role is only to listen for service requests from remote hosts. Once a request has been received, the server forks a child process to handle it, and itself goes back to listening for more requests. Because it allows new service requests to be accepted while previous ones are still being served, many programs, including the Internet Superserver (`inetd`), use the concurrent server model.

A typical program template for concurrent servers using sockets is shown in Figure 5. The program uses a socket `fd` to listen for incoming service requests. When a request arrives from a remote endpoint, it is accepted and a network connection is established. The local endpoint of the new connection is accessible through `newfd`. The server then forks a child process to serve the request and itself closes `newfd`.

In the caching mechanism described so far, `newfd` will be cached to the server on being returned by `accept()`. Immediately afterwards, however, the server does a `fork()` to serve the request in a child process, forcing `newfd` to be flushed in Migrating Sockets, because it is now shared between the server and the child. In fact, though, the server no longer needs access to `newfd`. When it closes `newfd`, the socket becomes cached to the child process.

Notice that although the final objective is to cache the accepted socket in the child process, two cache and one flush operations are involved, of which one cache and one flush would be unnecessary. To solve the problem, a new socket option called SO_CONCURRENT_SERVER is supported my Migrating Sockets. When the option is set for a socket, say S, in the listen state, it serves as a hint that sockets accepted through S should ultimately be cached to a child process forked by the process, say P, doing the listen. Hence, when a socket is accepted through S, it is merely marked *cacheable* instead of being cached to P. If later, P does access S for send/receive, S will be cached (this is known as *delayed caching*). If, however, P does a fork before it accesses S, S will be cached to the forked process, say Q, as part of copying file descriptors from P to Q. Moreover, S is marked "recall-on-access" in the network server, meaning that if P later accesses S, the network server will recall S from Q, in effect causing S to be flushed. However, it is more likely that P

will soon close S, and the recall-on-access status of S can be cleared.

4.3 Routing information management

The Internet protocol suite owes much of its flexibility and robustness in a heterogeneous and dynamic networking environment to protocols at (or below) the network layer. These protocols include, among others, IP, ICMP, IGMP, EGP and ARP. Together, they allow network routes to be dynamically discovered and reconfigured, and network connectivity is not lost even as network interfaces and routers come up or go down. In our system, we refer to the data structures that keep track of various kinds of routing information collectively as the *routing table*.

Dynamic routing interacts with caching of network endpoints. The reason is that even after a network connection has been established between a sender and receiver, the next "hop" (represented by an IP address) to which the sender must send its packets in order to reach the receiver may still change over time. This can happen, for example, by way of an ICMP redirect message, and the routing table must be updated to reflect the change. Moreover, the network interface to which some IP address has been assigned may be replaced by another interface, and ARP must update its translation of the IP address (in the routing table) to the link level address of the new interface.

In our design, we consider routing table management a global system function. As such, the network server is the only process responsible for its management. This has two advantages. First, routing table management functions do not have to be duplicated in the address space of every application process. Second, application processes do not need to be interrupted by (and process) routing messages that do not affect them. However, since application processes need read access to the routing table even for common case send and receive, such access must be as inexpensive as if the routing table were local to each process.

To satisfy the efficiency requirement, the network server creates a shared memory region, and allocates routing table entries exclusively from that region. Moreover, data pointers in the routing table must retain their intended meaning (without translation) irrespective of the process accessing them. This requires the network server and each application process to map[2] the shared memory region at a "well-known" virtual address. This virtual address can be returned by the network server to a client process at client registration time.

Our shared memory solution allows application processes to freely read the system routing table. We believe, however, that this does not represent a security problem in most cases. For example, users on a Unix system are often permitted to use the netstat(1) command to return the same kind of information.

4.4 Thread folding

Beyond the basic multi-threaded architecture described in Section 3.3 and illustrated in Figure 3, Migrating Sockets supports a mechanism called *thread folding*. The mechanism works as follows. Before the system schedules a receive protocol thread to read from a socket endpoint, it checks if there is already an upper-half application thread reading from the same endpoint. If so, the receive protocol thread will not be scheduled. Instead, when packets arrive, they will be directly delivered to the application thread, in whose context receive side protocol processing will be performed. If not, then the receive protocol thread will be scheduled as in the basic architecture; this is needed to handle reliable protocols such as TCP, where receive side ACK or exception processing is needed even in the absence of application reads.

Thread folding has two major benefits. First, when an application thread is already reading from the socket endpoint, we save a context switch from a receive protocol thread to the application thread. Second, when an application thread is actively processing network data (e.g., it is receiving continuous media), it is sufficient to determine a progress rate for the application thread alone, without having to do so for a protocol receive thread.

[2] This region is mapped read-only by application processes.

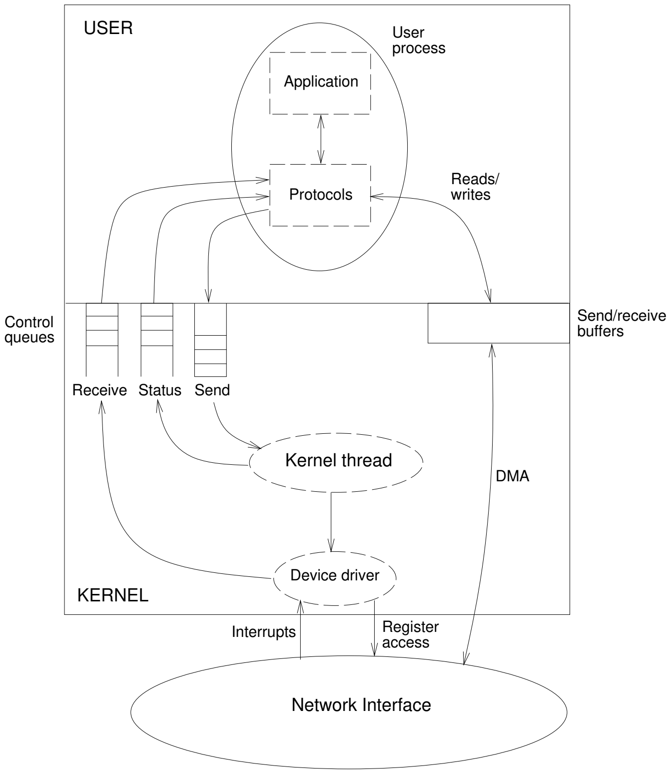

Figure 6: OS architecture for multimedia networking.

4.5 Protocol/kernel interface

For efficiency, Migrating Sockets runs on top of an OS architecture (Figure 6) we have previously prototyped for supporting continuous media (CM) applications [27].

Send/receive buffers shown in Figure 6 are allocated using the `IOBuffer::IOBuffer()` method in Table 1. The method creates a *network buffer* region for direct send/receive to/from the network (i.e. no intermediate data copies are required). If the network interface for send/receive uses DMA, the allocated network buffers will be automatically backed by required DMA resources. Moreover, buffers can be pinned in physical memory for predictable performance. A memory allocator associated with a buffer region supports flexible memory allocation/deallocation similar to `malloc(3C)` and `free(3C)` in the standard C library.

To send a packet, a user process appends control information in the form of a *send request* to a send control queue, managed through the `SendControlQueue` object in Table 1. The send request is then handled by the kernel through either a system call or a kernel thread introduced below. Notice that for some systems, even after kernel code processing of a send request has completed, the packet to send may only have been queued to a network interface, instead of really sent. Therefore, the buffer for sending cannot be reused or freed until the network interface has completed its part of the send and updated the status of the send request. That is why a method such as `reapall()` in Table 1 is necessary.

On the receive side, driver code for a network interface informs user processes of data to read by appending receive notifications to a receive control queue, managed through the `RecvControlQueue` object. For efficient control transfer, send/receive control queues are shared between user processes and the kernel.

A lightweight kernel thread provides shared access to a reserved-rate network connection in future integrated services networks. The kernel thread is periodically scheduled and implements a rate-based packet scheduling algorithm such that multiple user processes can send packets to the network connection with guaranteed data rates. For this purpose, a process creates a `MultiplexGroup` object using the

`MultiplexGroup(int, int, int, void *, int)` method shown in Table 1. The method causes a *multiplex group*, a data structure used by the kernel thread for rate-based packet scheduling, to be created within the kernel. The alg parameter specifies the packet scheduling algorithm to use for the multiplex group. Currently, the `KT_RC` algorithm in [27] is supported. Parameters of the scheduling algorithm can be passed with the params pointer. For example, the `KT_RC` algorithm takes the scheduling period (in μs) of the kernel thread as a parameter. Once a multiplex group (identified by a key which is unique within an end system) has been created by a process, other processes can gain access to the group using the `MultiplexGroup(int)` method. Processes having access to a multiplex group can use the `join()` method to add traffic flows to the group with specified parameters. A rate parameter (in kbps), for example, is needed for a flow using the `KT_RC` algorithm. Flows can be deleted from a multiplex group using the `leave()` method.

Lastly, notice that, although the interface presented in Table 1 is designed to be general and device independent, some of the operations accessed through the interface are device dependent. For example, the `SendControlQueue::send()` method calls a device specific send function within the kernel. The `IOBuffer::IOBuffer()` method creates device dependent DMA resources backing allocated network buffers.

5 ARC Scheduling of Protocol Threads

Application and protocol threads in Migrating Sockets can specify their CPU requirements using the rate-based reservation model of ARC scheduling [28]. The rate-based model has two parameters: (1) rate, r, ($0 < r \leq 1$), and (2) period, p, in μs. Informally, the rate specifies a guaranteed fraction of CPU time that a thread with the reservation will be allocated over time intervals determined by p.

Progress guarantees of rate-based reservations are provided by instances of a *family* of ARC schedulers with the following properties: (1) reserved rate can be negotiated, (2) QoS guarantees are conditional upon thread behavior, and (3) firewall protection between threads is provided. The first property is provided by a *monitoring module* and a *rate-adaptation interface* as discussed in [28]. The second and third properties are provided by using an on-line CPU scheduling algorithm with the *firewall property*, such as the RC scheduler in [28]. Subject to the admission control condition that the aggregate reserved rate does not exceed one, RC provides the following progress guarantee: a "punctual" thread with rate r and period p is guaranteed at least krp CPU time over time interval kp, for $k = 1, 2, \ldots$. Because ARC schedulers offer firewall protection between "well-behaved" and "greedy" threads, they are appropriate for integrated scheduling of continuous media and other applications found in a general purpose workstation. Besides RC, scheduling algorithms with improved fairness have also been deployed in ARC, such as the *fair rate-controlled* algorithm in [26].

ARC implements priority inheritance for threads that can contend for synchronization resources such as semaphores, mutex locks, and readers/writers locks. This reduces the extent of priority inversion, wherein a lower priority thread blocks a higher priority thread by holding a lock required by the latter. (When coupled with a dynamic priority ceiling protocol such as [10], the extent of priority inversion can be bounded.)

Recently, we have extended ARC to ARC-H (where H stands for heterogeneous services) to explicitly handle diverse classes of application requirements, including both guaranteed and best-effort services. An overview of ARC-H is given below (see [26] for a detailed treatment).

An ARC-H system administrator can partition the total CPU capacity into rates for m service classes, i.e., service class k is allocated rate R_k, $1 \leq k \leq m$, such that $R_k > 0$ and $\Sigma R_k = 1$. For $k = 1 \ldots m$, an overbooking parameter, b_k, ($0 \leq b_k \leq \infty$) is also specified.

Thread j can request from service class k a reservation specified by two parameters: *nominal rate* $\hat{r}_j$ and period p_j. The request is granted if

$$\Sigma_{i \in C_k} \hat{r}_i + \hat{r}_j \leq R_k(1 + b_k),$$

where C_k denotes the subset of threads already admitted into service class k.

After thread j has been admitted, it receives an *effective rate* given by

$$r_j = R_k \times (\hat{r}_j / \Sigma_{i \in C_k} \hat{r}_i),$$

where C_k is the subset of threads admitted into service class k, which by now includes thread j. These effective rates, $r_j, j = 1 \ldots n$ (n is the total number of threads) in ARC-H have the same interpretation as the reserved rates in ARC scheduling [28].

The overbooking parameters are used for specifying different levels of service. For $b_k = 0$, threads in service class k are provided with a hard guarantee of their reserved rates. For $b_k = \infty$, service class k can be used for flexible rate allocation with excellent scalability (but threads in this class receive no guarantee at all). Other values of b_k lead to service classes with a statistical guarantee of different strengths.

Note that ARC-H provides firewall protection between service classes such that service class k receives a hard guarantee of reserved rate R_k, for all k.

6 Active Demultiplexing

An important task of protocol processing is to demultiplex incoming packets to their network endpoints. Traditionally, the receive side of a kernel level transport protocol looks for matches by searching a list of protocol control blocks known to the system.

Recent user level protocol implementations have relied on packet filters installed with the driver of a network interface to accept or reject packets. While highly flexible, these methods involve searching. Although techniques such as hashing and one behind cache can significantly reduce the search time on the average, the actual search time may still be highly variable when the number of network endpoints in a system is large.

Since predictable performance is an important goal of our architecture, our system supports a constant overhead packet demultiplexing mechanism known as *active demultiplexing*. The basic idea is that, under certain conditions, an OS handle identifying a receive process can be included in packets destined for that process. An end system can then make use of the OS handle to deliver packets directly to the receive process, without any searching.

Currently, active demultiplexing exploits the notion of exclusive packet receivers introduced in section 3. In situations where active demultiplexing cannot be applied (such as multicast) or is not preferred, our system provides packet filters.

6.1 Mechanism

We have implemented active demultiplexing in the context of TCP/IP. The mechanism can be transparently enabled when both sender and receiver hosts in a TCP connection support it. Figure 7 illustrates the mechanism, triggered when a socket, say S, that is an exclusive packet receiver becomes cached to a user process. An OS handle for the user process to receive from S becomes known as part of the process of caching. The newly cached socket S *advertises* the OS handle when it next sends a packet to the remote endpoint, say R, of the network connection. This happens when either S has data to send to R, or when S acknowledges packets received from R. The advertisement is carried in a new TCP option called TCPOPT_DEMUX_ADVERTISE.

On processing TCPOPT_DEMUX_ADVERTISE, R learns about S's OS handle. R then caches the handle and can later *enclose* it in subsequent packets it sends to S, by way of a new IP option IPOPT_DEMUX_ENCLOSED. The enclosed OS handle enables active demultiplexing by S's end system. To allow a link level device driver to easily locate IPOPT_DEMUX_ENCLOSED, the option is always inserted as a first IP option. In addition, we use a currently unused bit in the service field of IP header to indicate to the device driver whether an OS handle has been enclosed.

An OS handle should be revoked when a cached socket becomes no longer exclusive, or when an exclusive socket becomes flushed. Handle revocation is achieved using a new TCP option TCPOPT_DEMUX_REVOKE. On receiving a handle revocation from its peer, a network endpoint stops including IPOPT_DEMUX_ENCLOSED in packets sent to the peer. Figure 8 shows the formats of the various options used for active demultiplexing. Notice that a sequence number is included in handle advertisements and revocations. We prescribe that the

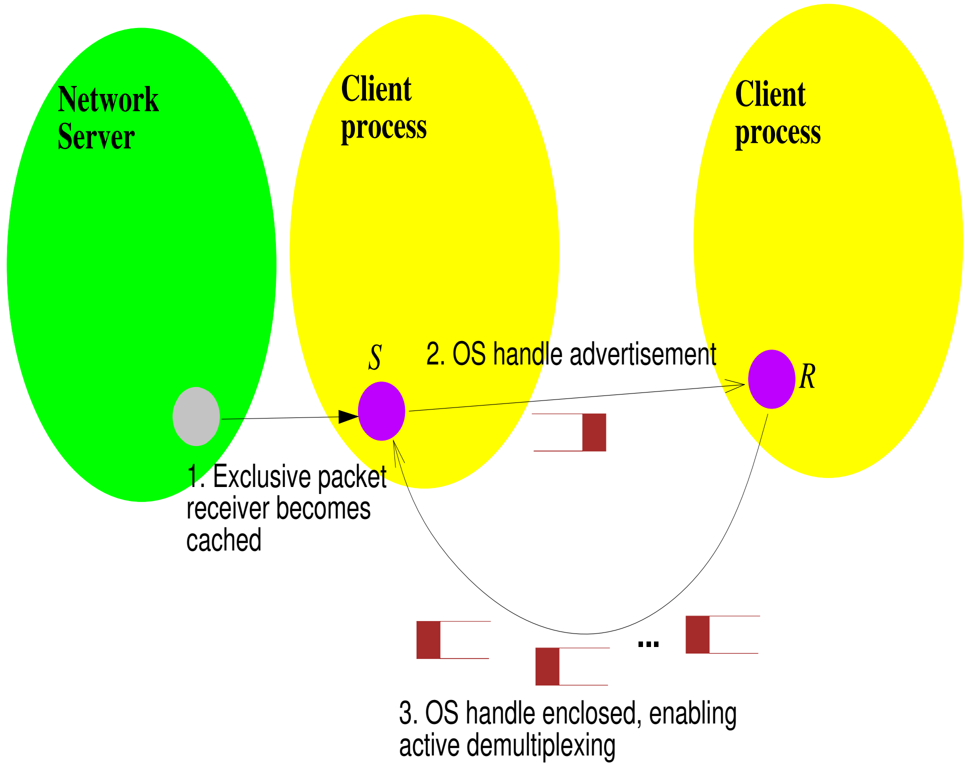

Figure 7: Mechanism of active demultiplexing for TCP/IP.

two operations be applied only in increasing sequence number order, thereby preventing an earlier operation from overwriting a more recent one.

The complete state transition diagram for handle advertisement and revocation using TCP/IP is shown in Figure 9. If (and only if) an endpoint is in the “advertising” state, the advertisement option will be used for TCP segments sent by the endpoint. Similarly, if (and only if) an endpoint is in the “revoking” state, the revocation option will be used for TCP segments sent by the endpoint.

Notice from Figure 9 that handle advertisement for some network endpoint, say S, should persist until one of two conditions occurs. First, when new (i.e. non-retransmitted) TCP data have been sent and acked for S, in which case a data packet carrying the handle advertisement in question will have been reliably delivered. Hence, a transition from the advertising to the advertised state occurs. Second, suppose n is the highest receive sequence number acked by S when the advertising state in Figure 9 was last entered. Subsequently when TCP data with sequence number higher than n were received for S, a TCP ack advancing S's receive window must have been successfully delivered to S's peer endpoint. Moreover, the ack must have been carried in a TCP segment containing the handle advertisement in question. Hence, a transition from the advertising to the advertised state occurs. Similar conditions govern the transition from the revoking to the revoked state.

Our experience has been that it is straightforward to incorporate active demultiplexing into TCP/IP. To give a rough idea of the complexity of our implementation, we report that 110 lines of C code were added to TCP input processing, 22 lines were added to TCP output processing, 51 lines were added to the socket layer, and 15 lines were added to the link level device driver.

Option name	Option length	Sequence number	nonce	OS handle

TCP handle advertisement

Option name	Option length	Sequence number

TCP handle revocation

Option name	Option length	nonce	OS handle

IP enclose handle option

Figure 8: Formats of active demultiplexing options.

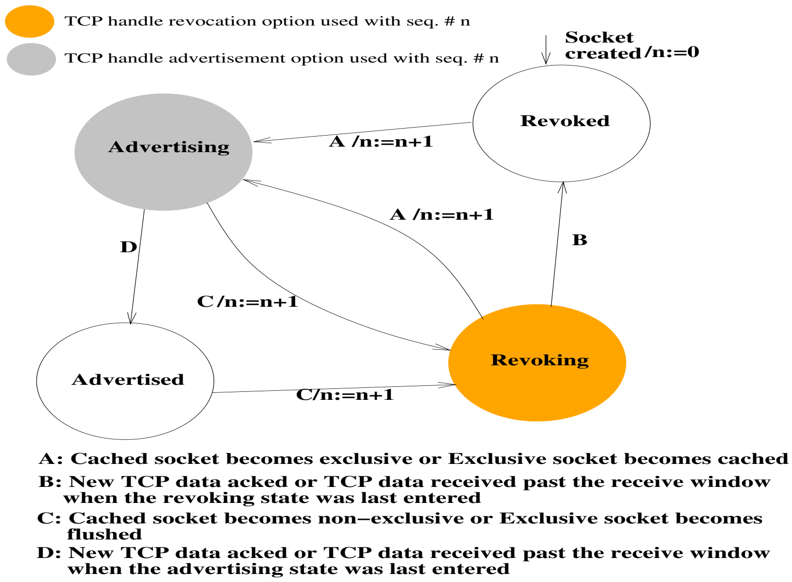

Figure 9: State transition diagram for handle advertisement and revocation in TCP/IP.

6.2 Security considerations

Taking an OS handle in an incoming packet and using it to deliver the packet directly to a receive process is a “powerful” mechanism. Without proper precautions, the mechanism can raise some serious security concerns. First, a malicious process can advertise an indiscriminate OS handle (i.e. one that is not associated with an exclusive packet receiver) and “hoard” packets that should also be delivered to other processes in the system. Second, an OS handle can become obsolete, such as when a cached endpoint becomes flushed. Third, a faulty process can enclose a wrong or fabricated OS handle in sending packets.

To guard against such security problems, we generate a *nonce* when a network endpoint is installed and associate it with the endpoint. The nonce is used in conjunction with the OS handle for active demultiplexing. Nonces have the properties that each newly generated nonce has a fresh value, and it is difficult to guess the

Algorithm ACTIVE

begin
1. Remember enough execution state to transfer to fallback on data fault;
2. **if** (handle is not for an exclusive packet receiver)
   goto fallback;
3. **if** (nonce in handle does not match nonce for receive endpoint)
   goto fallback;
4. Deliver packet directly to process identified in handle;
5. **return**;

fallback:

6. Match packet against installed packet filters in system;

end ;

Figure 10: Specification of active demultiplexing algorithm.

value of a nonce that is not explicitly passed. Before accepting an OS handle, receive side demultiplexing code performs security checks as shown in algorithm `ACTIVE` (Figure 10). Notice that the check at line 2 guards against the first security threat in the preceding paragraph, while the checks at lines 1 and 3 guard against the second and third security threats. We fall back on a conventional packet filter mechanism if an enclosed OS handle is found unacceptable.

Finally, we note that, in practice, security can be better enforced if an OS handle is inserted by kernel level code (i.e. by a link level device driver supporting active demultiplexing). In this case, security safeguard is mainly to limit the cycle time of nonces, and it is sufficient to generate a new nonce by incrementing it cyclically.

7 Experimental Results

We have an implementation of Migrating Sockets on Solaris 2.5.1. We are currently running it on Sun SPARC/UltraSPARC workstations and Pentium II desktops interconnected by 10/100 Mbps Ethernet networks. Major Internet protocols including TCP, UDP, and IP (with IP multicast) have been ported from 4.4 BSD to Migrating Sockets. In the following subsections, we first present experimental results on the overheads of individual system components in our prototype. We then present results on the overall system performance and the effectiveness of our QoS support mechanisms.

7.1 Component costs

Thread synchronization

Migrating Sockets makes use of a multi-threaded programming model. The model has the inherent cost that protocol threads have to synchronize with each other using mutex locks and condition variables. Moreover, context switches are required to switch execution between threads. In our first set of experiments, we measure various thread synchronization overheads. The purpose is to show that the use of multi-threading does not result in excessive overhead. For our measurements, we performed the operation in question many times between two threads, and report the average time taken.

The first row in Table 2 shows the times needed to acquire and release a mutex lock for the different SPARC architectures. The second row shows the costs of signaling a condition variable. The third row shows the context switch times from one thread to another. In our current system, usually two mutex locks are needed for sending a packet, while one mutex lock and one condition signal are needed for receiving a packet. Context switching is mainly required on the receive side, from a receive thread to an application thread. Its cost can often be amortized over a train of packets received. From Table 2, we conclude that the cost of thread synchronization is an acceptable fraction of the total cost of protocol processing (see Tables 3 and 4).

TCP/UDP performance

We measured the performance of our implementation of TCP/IP and UDP/IP protocol stacks accessed through a cached socket in Migrating Sockets. The objectives are (1) to demonstrate that real and complex internetwork protocol stacks can be effectively implemented in our framework, and (2) to identify where major protocol processing time is spent in an end-to-end communication, thereby identifying further opportunities for performance improvement. For UDP, we have incorporated the following optimization techniques proposed in [19]: (1) Integrated checksumming and copying of data from application buffers to network buffers, (2) replacement of general purpose socket send code with more efficient UDP specific code, and (3) deletion of pseudo-connect in UDP send. Moreover, since Internet checksumming is heavily used, we replaced the 4.4 BSD checksum routine with a more efficient routine optimized for the SPARC Ultra-1 architecture. Apart from optimized checksumming, the TCP code is largely ported as is from 4.4 BSD.

To do the measurements, we used the Solaris TNF facility to insert *probe points* at strategic places of the code. An executed probe point logs, among other things, a timestamp useful for timing analysis. We present results averaged over a large number of data points taken. However, the performance numbers are subject to the overhead of TNF probe points executed at the user level. We believe they are useful for comparing the relative costs of components in our system.

Table 3 gives a breakdown of the various (average) component costs (the numbers in brackets are corresponding standard deviations) due to host software on the send path of a packet. Each packet carried either 1400 bytes or one byte of user data. The relatively high costs of socket and TCP send code for 1400 bytes were mainly due to data copy and TCP checksumming. For UDP, socket send code had minimal cost because it called a UDP specific send function very early on. The UDP function performed integrated checksum and copy of data from application to network buffers. Performance benefits of replacing "baroque" socket send code with protocol specific code can also be seen by comparing the costs of socket send for one byte of TCP and UDP data, respectively.

The row labeled "Link layer" in Table 3 gives the cost of link layer processing such as ARP address translation by Migrating Sockets. Also notice from the last row of the table that the kernel level overhead, i.e., processing by the network device driver, was largely insensitive to the packet size. This is because protocol code allocated network buffers for sending and data did not have to be copied from user to kernel space.

We conclude that, on the send path, major processing time is spent at the transport level (TCP or UDP). Socket level processing can also be expensive, but can be reduced by replacing general purpose socket code by more efficient protocol specific code.

Table 4 gives a breakdown of the various (average) component costs (the numbers in brackets are the corresponding standard deviations) due to host software on a TCP/IP or UDP/IP packet receive path. As for the send path, a packet carried either 1400 bytes or one byte of user data. The "Kernel interrupt" number is the total time spent in the interrupt handler of the receive network device driver, which includes the cost of matching a packet to a BSD packet filter (which took roughly 10 μs). A matched packet woke up a receive thread of Migrating Sockets and caused the thread to be scheduled. The cost of context switching to the receive thread is given by the "Switch to recv thread" number.

Protocol processing of a received packet by Migrating Sockets starts at the link layer. The "Link layer" number includes costs such as inspecting the type field of the Ethernet header and dispatching the packet to IP. TCP or UDP processing of a packet took longer for 1400 bytes than for one byte of user data, due to checksumming of a longer packet. After the transport layer, the receive thread appended the packet to a

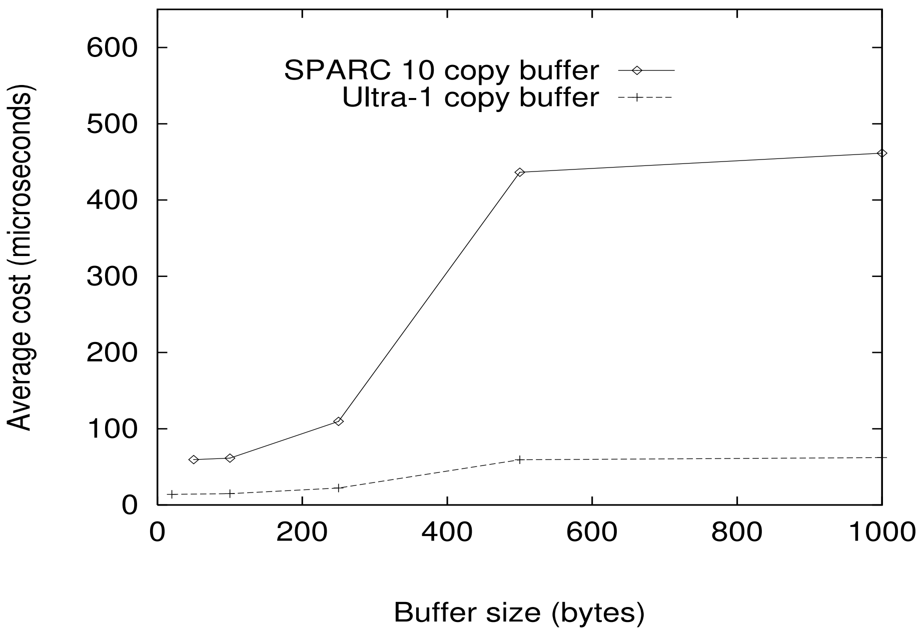

Figure 11: Copy buffer overhead (in μs) of buffer management subsystem (Ultra-1 and SPARC 10).

socket receive buffer and woke up an upper-half application thread that was blocked reading from the socket. The "Switch to read thread" number gives the overhead of context switching to the application thread. Lastly, the "Return from read" number includes the cost of copying any user data from a socket receive buffer to an application buffer. Hence, it was higher for a larger packet size.

We note that, as is the case with the send path, transport level processing is an important source of overhead on the receive path. However, the receive path is more involved because of two needed context switches (assuming no thread folding) – one from kernel interrupt handling to a protocol receive thread, the other from the protocol receive thread to an application thread performing a read.

Protocol runtime support

Migrating Sockets implements a buffer management subsystem and timer subsystem different from those in 4.4 BSD. We report on the performance of these subsystems. The purpose is to give an idea of how they might impact perfomance when used to support protocol stacks other than TCP/IP and UDP/IP.

We measured the performance of the buffer management subsystem in our current implementation. Figure 11 shows the overhead for *copying* a *message block*. The operation consists of allocating a message block and copying buffer data from an existing block to the new one. The size of the message block was varied in the experiment. Figure 12 shows the overhead for *duplicating* a message block. The duplicate operation is similar to the copy operation, except that buffer data are not really copied, but are reference counted for sharing. It is very useful when protocol layers need to share data without modifying the data. The cost of the duplicate operation is substantially lower than that of message block copy and is mostly independent of the buffer size (Figures 11 and 12).

In protocol processing, it is frequently necessary to *link* two message blocks together, such as when a header is to be prepended to an existing packet. Figure 13 gives the overhead of this operation. The overhead is quite small since it involves only simple pointer manipulations.

We next report on the performance of the timer subsystem. Figure 14 shows the time needed to insert/delete a timer into/from the timeout table. The times to timer expirations were randomly chosen from zero to 500 seconds. The total number of timers used in an experiment was varied. Figure 15 shows the time needed

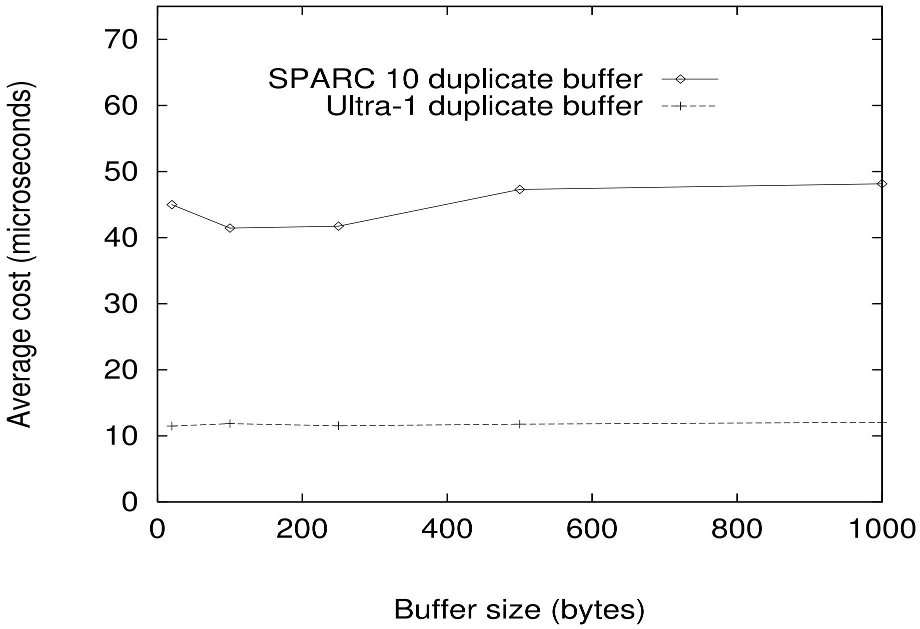

Figure 12: Duplicate buffer overhead (in μs) of buffer management subsystem (Ultra-1 and SPARC 10).

to execute a null timer function for different numbers of timers used in an experiment. The times to timer expiration were randomly chosen from zero to 250 seconds.

Optimized checksumming

This set of experiments quantifies the performance benefits of using an Internet checksum routine that is optimized for a specific computer architecture, namely the SPARC Ultra-1. Figure 16 compares the performance between the original 4.4 BSD checksum routine and an optimized checksum routine for various data sizes. By exploiting knowledge of the most efficient data size and alignment for memory access, the optimized routine achieves significant improvement over the original one.

We next investigate the performance benefits of integrated checksum and copy. In the experiments, the destination address of a copy was double-word aligned. For the separate checksum and copy approach, the copy loop was done immediately after the checksum loop. Figure 17 shows the results when the test program had a small memory footprint (less than 8 M-Bytes). For the integrated approach, the performance was very slightly better if the source address was also double-word aligned than if it was word or byte-aligned. Also, the integrated approach showed minimal improvement over the separate approach because of cache effects. Figure 18 shows the results when the test program had a large memory footprint (41 M-Bytes). In this case, the integrated approach was significantly more efficient than the separate approach.

7.2 Performance impact

Comparison with Solaris

Although high performance is not the main concern in our work, we show that our system is competitive with an in-kernel protocol stack in Solaris, a mature and industry strength operating system. Competitive performance is achieved in part by our efficient protocol/kernel interface, which imposes little additional overhead on network access from the user level. Further optimizations with Migrating Sockets are possible,

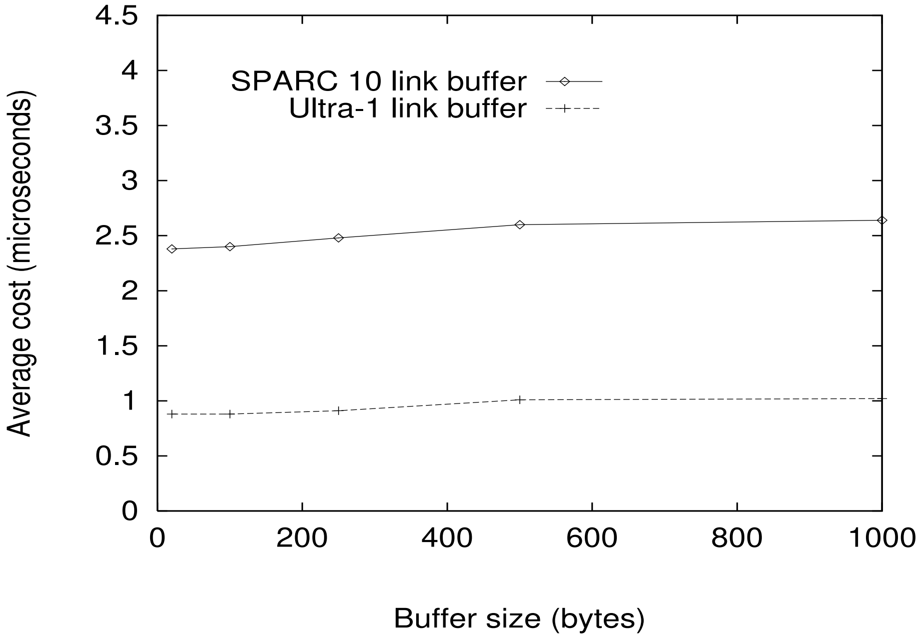

Figure 13: Link buffer overhead (in μs) of buffer management subsystem (Ultra-1 and SPARC 10).

chiefly by changing the sockets API to further reduce data movement,[3] and by implementing integrated layer processing.

In our experiment, we sent 8000 TCP/IP packets on a round trip between two Pentium II/300 machines connected by 10 Mbps Ethernet, using unmodified Solaris 2.5.1 and Migrating Sockets. We measured the average time taken for one round trip. As in Section 7.1, we used an application payload of one and 1400 bytes, for a small and large Ethernet packet, respectively. For one byte, Migrating Sockets and Solaris used 727 and 719 μs, respectively.[4] Hence, Migrating Sockets had an extra 8 μs overhead. For 1400 bytes, Migrating Sockets and Solaris used 4251 and 4174, μs, respectively. Hence, Migrating Sockets had an extra 77 μs overhead. We conclude that even without using aggressive optimization techniques (that are possible with user level implementation), Migrating Sockets achieved highly competitive performance.

Delay performance

To quantify the delay performance of ARC scheduled protocol threads in Migrating Sockets, we sent 2000 TCP/IP packets (with 1400 byte application payload) on a round trip between two socket applications running on two Ultra-1s. The time taken for each round trip was recorded. For competing workload, seven compute-intensive greedy applications ran on each machine during the course of measurement.

The experiment was repeated two times. In the first run, we ran both socket applications and all 14 competing applications in the Solaris TS class. Figure 19a plots the sequence of round trip times obtained. As shown in the Figure, interference from the greedy applications caused occasional but substantial increases in the measured round trip times.

In the second run, each socket application using Migrating Sockets ran with an ARC rate of 0.1, and each greedy application ran with a rate of 0.02. Figure 19b plots the sequence of round trip times obtained. Unlike

[3]For example,it is shown in [8] that solely by changing socket applications to directly allocate buffers from the network memory pool, bandwidth can be roughly doubled and latency can be roughly halved. In our system, the Migrating Sockets layer does directly allocate network buffers. However, we did not go one step further to do the same at the application level, because of our objective to remain backward compatible with Berkeley sockets and existing applications.

[4]These numbers include inherent latency due to interface hardware and restricted Ethernet bandwidth.

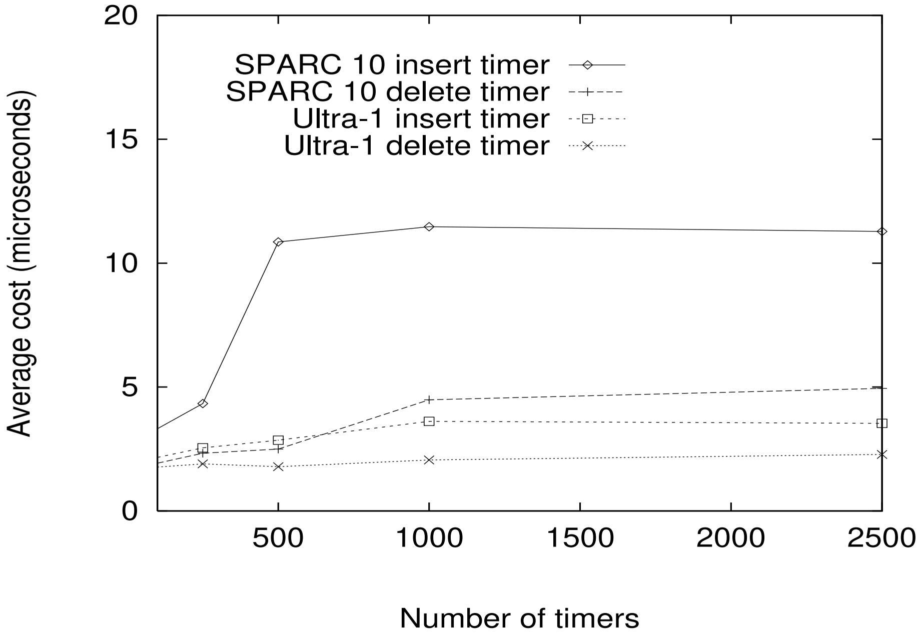

Figure 14: Insert and delete timer overhead (in μs) of timer subsystem (Ultra-1 and SPARC 10).

Figure 19a, the measured times in this case closely match those achieved when both socket applications ran standalone. Hence, protocol processing rates are guaranteed using Migrating Sockets.

Active demultiplexing

To quantify the performance impact of active demultiplexing relative to packet filters, we sent 2000 TCP/IP packets on a round trip between a pair of Pentium II/300 machines connected by 10 Mbps Ethernet, and measured the average time for one round trip. An application payload of 1400 bytes was used for each packet. Figure 20 shows the average time per round trip for active demultiplexing versus linear matching with packet filters, as we varied the number of receive endpoints on one of the hosts. As shown, active demultiplexing has superior performance when the number of endpoints becomes large.

However, by increasing the amount of control information in a packet, active demultiplexing represents a tradeoff between bandwidth and processor efficiencies. Our implementation uses twelve bytes for an enclosed OS handle. For 10 Mbps Ethernet with a maximum frame size of 1514 bytes, this represents a 9.6 μs increase in link processing time per packet, and a 0.7% loss of efficiency in link bandwidth. Increase in link processing time will decrease proportionately with higher bandwidth networks, whereas loss of bandwidth efficiency will decrease with larger link packets (such as FDDI and ATM AAL data units [23]).

Multimedia

This experiment was performed between two Ultra-1s connected by 10 Mbps Ethernet. We used a video client application, `mpeg2client`, modified from the public domain `mpeg2play` to read video data from the network. The application, using Migrating Sockets, repeatedly reads MPEG-2 encoded pictures carried in TCP/IP packets. After reading each whole picture, it decodes the video into a frame of 8-bit pixels, and records a timestamp. In our experiment, video was sent at 30 fps to `mpeg2client` by another application running on a different machine. The video was a 65 second segment of IPPPP encoded tennis instruction.

We ran `mpeg2client` with an ARC rate of 0.3 and period 33 ms. As competing workload, eight compute-intensive greedy applications ran with a rate of 0.02 each. A frame rate of 29.99 per second was

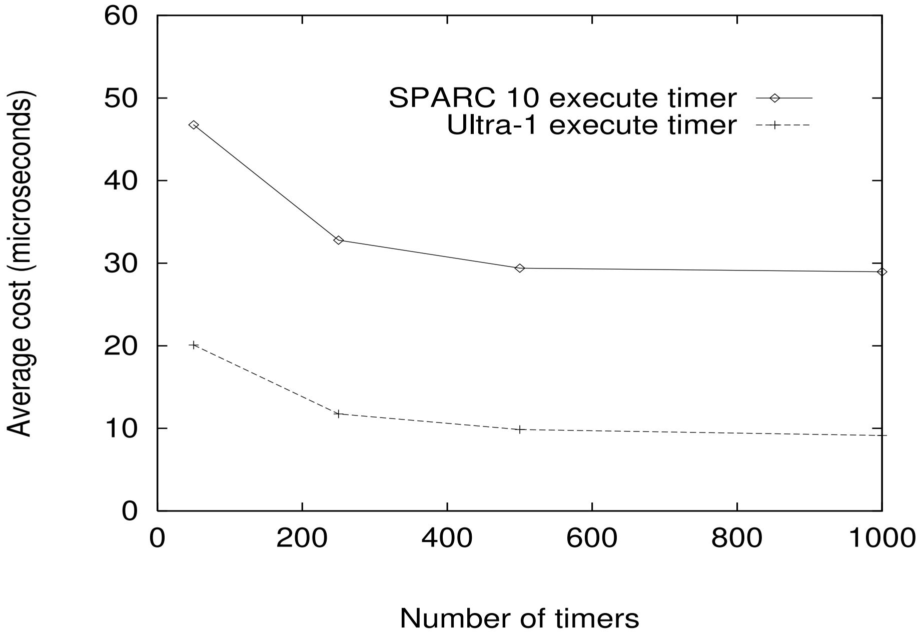

Figure 15: Execute timer overhead (in μs) of timer subsystem (Ultra-1 and SPARC 10).

reported by `mpeg2client` in the experiment, compared with a frame rate of 14 per second when all the applications ran in Solaris TS. Hence, `mpeg2client` received sufficient CPU time for full frame rate, despite the presence of competing workload. Moreover, Figure 21 shows a plot of the times between pictures decoded by `mpeg2client`. As shown, the scheduling jitters were such that these times never exceeded 66 ms, which is the worst case predicted by ARC scheduling for a scheduling period of 33 ms.

8 Conclusion

We presented Migrating Sockets as a framework for user level protocol implementation, and discussed its relations to other system components in an end system architecture designed to support networking with QoS guarantees. We discussed implementation experience and presented experimental results to show the performance and QoS properties of our current system.

Acknowledgment

The authors wish to thank Long Ma for helping with performance measurements of the system, and the anonymous referees for detailed suggestions to improve the quality of this paper.

References

[1] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proc. First Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[2] S. Bradner and A. Mankin. The recommendation for the IP next generation protocol. Internet RFC 1752, January 1995.

[3] Torsten Braun and Christophe Diot. Protocol implementation using integrated layer processing. In *Proc. ACM SIGCOMM '95*, Boston, MA, August 1995.

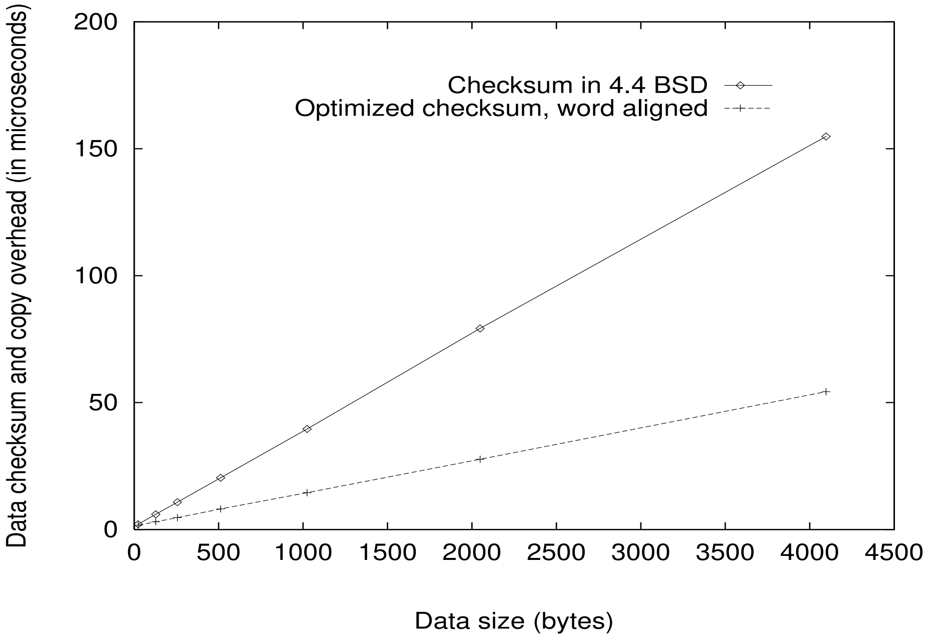

Figure 16: Checksum overhead (in μs) versus data size (in bytes) for Ultra-1.

[4] DEC Systems Research Center. Autonet-ii homepage. http://www.research.digital.com/SRC/home.html.

[5] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network Magazine*, 7(4):36–43, July 1993.

[6] D.R. Engler and M.F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM '96*, Stanford, CA, August 1996.

[7] ATM Forum. ATM traffic management specification, version 4.0, 1995.

[8] R. Gopalakrishnan and G.M. Parulkar. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[9] N.C. Hutchinson, S. Mishra, L.L. Peterson, and V.T. Thomas. Tools for implementing network protocols. *Software – Practice and Experience*, 1989.

[10] Min ih Chen and Kwei jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990.

[11] Van Jacobson. LBL whiteboard. Lawrence Berkeley Lab, software on-line at ftp://ftp.ee.lbl.gov/conferencing/wb.

[12] Van Jacobson. Visual audio tool. Lawrence Berkeley Lab, software on-line at ftp://ftp.ee.lbl.gov/conferencing/vat.

[13] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated CPU and network IO QoS management in an endsystem. In *Proc. 7th International Workshop on Quality of Service (IWQoS 97)*, 1997.

[14] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in Real-time Mach. In *Proc. Multimedia Japan*, April 1996.

[15] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol in Real-time Mach. In *Proc. IEEE Real-time Technology and Applications Symposium*, June 1996.

[16] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. 14th SOSP*, pages 244–255, December 1993.

[17] S. McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, CA, Winter 1993.

[18] S. McCanne and Van Jacobson. vic: A flexible framework for packet video. In *Proc. ACM Multimedia '95*, 1995.

[19] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.

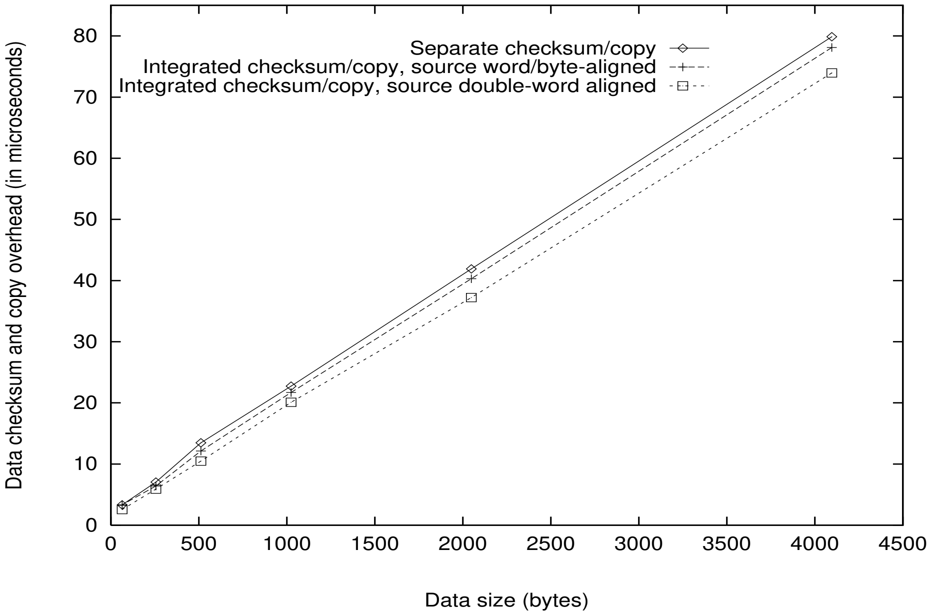

Figure 17: Data checksum and copy overhead (in μs) versus data size (in bytes) for small memory footprint (Ultra-1).

[20] K.K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser, and W. Duso. Operating system support for a video-on-demand service. *Multimedia Systems*, 1995(3):53–65, 1995.

[21] D.M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[22] Richard Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[23] Toshikazu Suzuki. ATM adaptation layer protocol. In *IEEE Communications Magazine*, pages 80–83, April 1994.

[24] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Networking*, 1(5):554–565, October 1993.

[25] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. 15th SOSP*, November 1995.

[26] David K.Y. Yau. ARC-H: Integrated CPU scheduling for heterogeneous services. Technical Report TR-98-024, Purdue University, West Lafayette, IN, July 1998. Revised.

[27] David K.Y. Yau and Simon S. Lam. An architecture towards efficient OS support for distributed multimedia. In *Proc. IS&T/SPIE Multimedia Computing and Networking*, pages 424–435, San Jose, CA, January 1996.

[28] David K.Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4), August 1997.

[29] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.

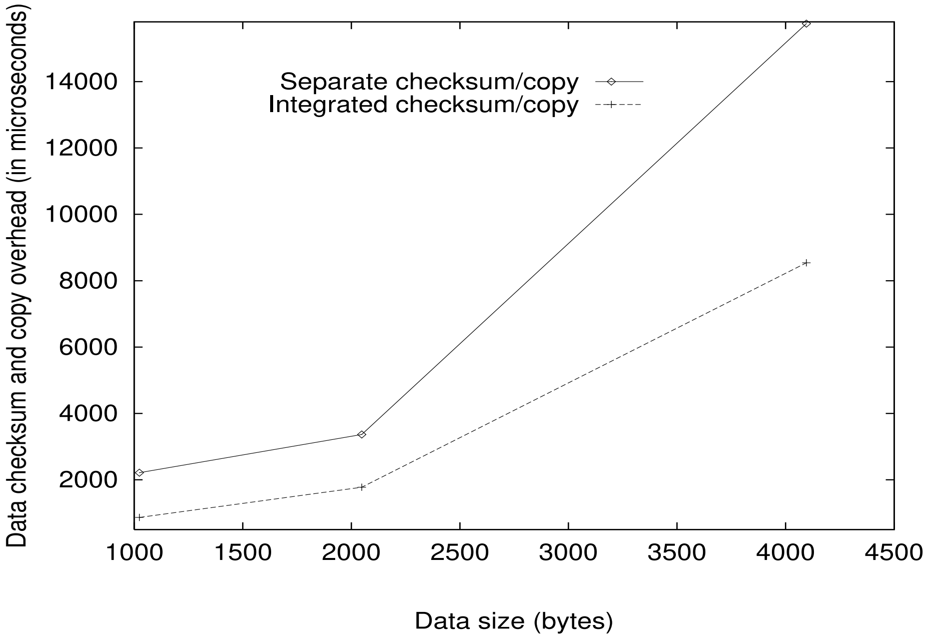

Figure 18: Data checksum and copy overhead (in μs) versus data size (in bytes) for large memory footprint (Ultra-1).

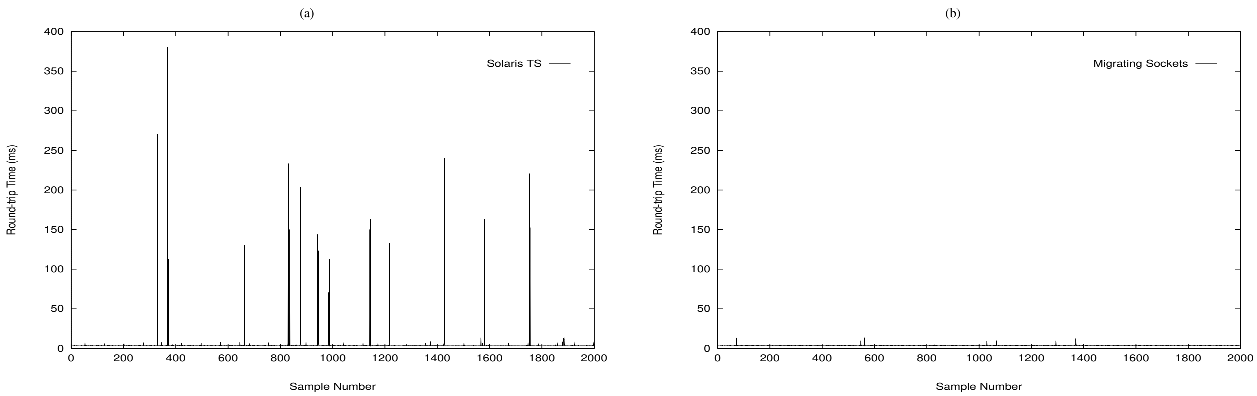

Figure 19: Profile of packet round-trip times with competing workload for (a) Solaris TS, and (b) ARC-scheduled Migrating Sockets.

Class	Method	Synopsis
IOBuffer	IOBuffer(int fd, caddr_t addr, int size);	Create a network buffer region of size bytes and map the region at user address addr. fd is a file descriptor for the network interface for which the buffer region is being allocated.
	void *malloc(int size);	Allocate a properly aligned buffer of size bytes from network buffer region.
	void free(void *buf);	Free to network buffer region buffer buf previously allocated by malloc().
SendControlQueue	SendControlQueue(int fd, caddr_t addr, int size);	Allocate a send control queue of size send notifications for the network interface identified in fd. The send control queue is to be mapped at user address addr.
	void attach(IOBuffer *iob);	Attach network buffer region iob to send control queue.
	int send(void *buf, int len);	Send len bytes starting at user address buf by appending a send notification to send control queue.
	struct status *reap(int block);	Return pointer to status of last newly completed send.
	void reapall();	Free buffers in all newly completed sends by calling free() method of attached IOBuffer.
RecvControlQueue	RecvControlQueue(int fd, caddr_t addr, int size);	Allocate a receive control queue of size receive notifications and map it at user address addr. fd is a file descriptor for the network interface for which control queue is allocated.
	int recv(caddr_t *buf, int *len, int block);	Receive len bytes of data in network buffer whose pointer is returned in buf. If no data are available for receive, block calling thread iff block is set.
	void unmap(caddr_t buf);	Give back buffer buf previously returned by recv() to kernel.
MultiplexGroup	MultiplexGroup(int key, int bw, int alg, void *params, int perm);	Create multiplex group with key key for a reserved rate network connection, with bandwidth bw Mbps. Multiplexing algorithm is specified by alg, with algorithm parameters pointed to by params. Permission for other processes to access multiplex group is specified in perm.
	MultiplexGroup(int key);	Get a multiplex group previously created with key key.
	int setoption(int optname, void *optval);	Set option for multiplex group. Option name is optname and value is pointed to by optval.
	int join(void *params);	Let a flow join multiplex group with parameters pointed to by params. Return id for the flow in multiplex group.
	void leave(int id);	Delete flow with id id from multiplex group.

Table 1: Kernel interface for protocol code in Migrating Sockets.

Overhead	Machine		
	SPARC 10	SPARC 20	Ultra-1
mutex lock	2.0	1.4	0.9
condition signal	0.5	0.4	0.2
context switch	47.0	32.5	18.0

Table 2: Thread synchronization overhead (in μs) in Solaris for various SPARC architectures.

Code	TCP send		UDP send	
	1400 bytes	1 byte	1400 bytes	1 byte
Socket send	90(74)	72(62)	16(9)	15(9)
TCP/UDP	144(73)	25(4)	101(62)	28(7)
IP	10(8)	10(2)	13(3)	13(3)
Link layer	9(2)	9(2)	15(3)	14(3)
Device driver	20	19	20	20
Total	273	135	165	90

Table 3: Breakdown of Migrating Sockets TCP/UDP send path latency (μs) for Ultra-1.

Code	TCP receive		UDP receive	
	1400 bytes	1 byte	1400 bytes	1 byte
Return from read	71(28)	52(24)	45(24)	30(11)
Switch to read thread	79(16)	92(51)	56(18)	56(16)
TCP/UDP	90	56	89	57
IP	15	14	9(3)	10(3)
Link layer	8(7)	7(5)	6(3)	7(3)
Switch to recv thread	34	33	38	32
Kernel interrupt	33	32	33	31
Total	330	286	276	223

Table 4: Breakdown of Migrating Sockets TCP/UDP receive path latency (μs) for Ultra-1.

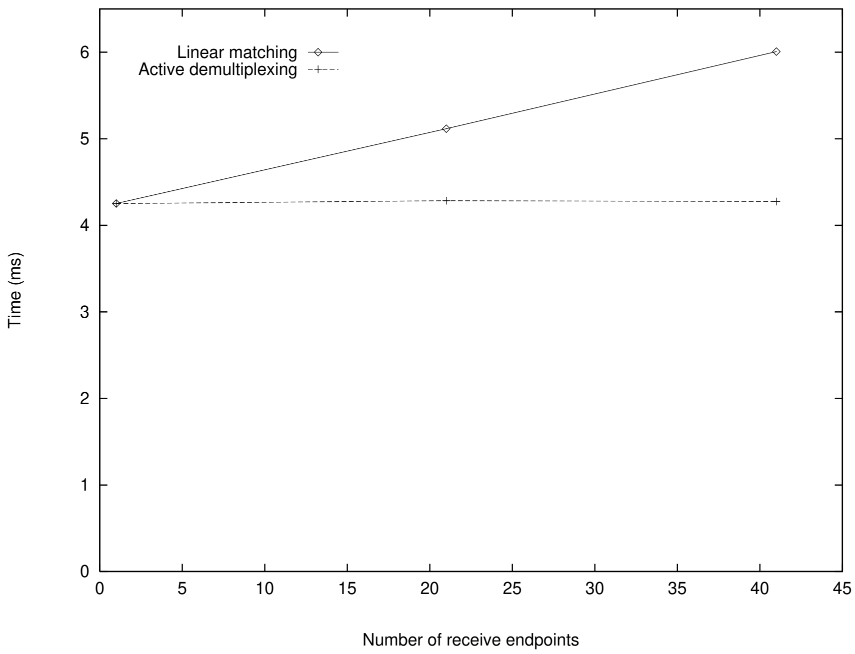

Figure 20: Performance impact of active demultiplexing vs linear matching, as a function of the number of receive endpoints.

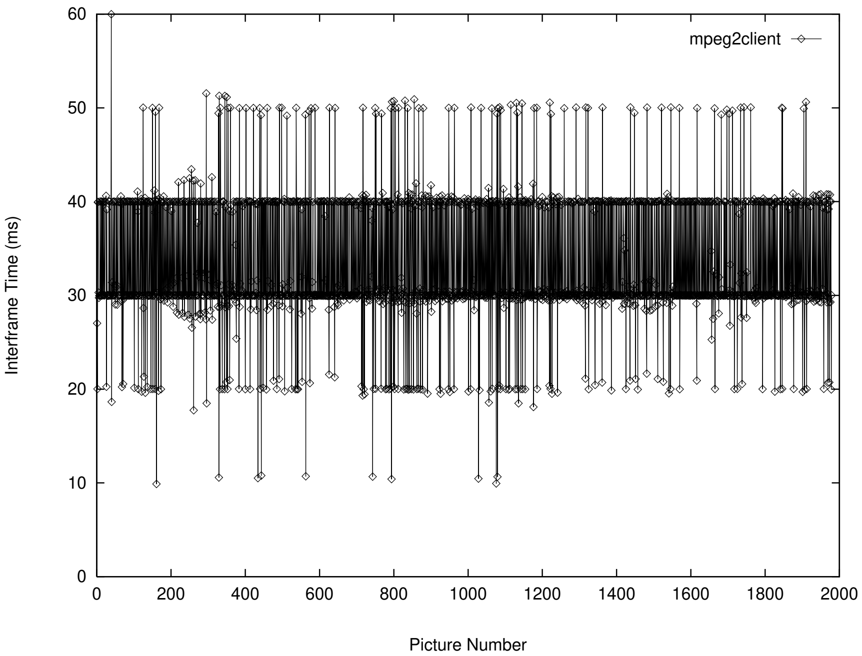

Figure 21: Times between pictures decoded by `mpeg2client` at 30 fps.