Experiences implementing a high performance TCP in user-space

Aled Edwards, Steve Muir*

Hewlett-Packard Laboratories, Filton Rd, Bristol, UK

Abstract

The advantages of user-space protocols are well-known, but implementations often exhibit poor performance. This paper describes a user-space TCP implementation that outperforms a 'normal' kernel TCP and that achieves 80% of the performance of a 'single-copy' TCP. Throughput of 160 Mbit/s has been measured. We describe some of the techniques we used and some of the problems we encountered.

1 Introduction

Network protocols are typically implemented in-kernel for reasons of performance and security. However, in-kernel implementations are often hard to debug and difficult to customise. Also, kernel protocols sometimes obstruct the development of application-specific protocols, as the kernel code is difficult to circumvent. While these problems are well-known, attempts at providing user-level protocols have often exhibited poor performance. Possible culprits for this poor performance are extra context switches and system calls. As performance is a paramount consideration user-space protocols are rarely found in 'real-world' systems.

This paper outlines some techniques to avoid these problems and describes the implementation of a complete TCP in user-space that exhibits very high performance. The work was carried out on an experimental Gbit/s network called Jetstream.

We start by describing the Jetstream network and the low-level interface we added to its network device driver to support development of application layer protocols. The implementation of TCP in user-space using these facilities is then discussed. We describe problems and pitfalls we found providing a user-space TCP in a Unix environment and point out features of TCP that made the implementation difficult. Finally, we present results that show our user-space implementation outperforms the normal kernel TCP, and discuss optimisations we plan to implement in future work. We conclude that user-space protocol implementations can perform well given the appropriate low-level driver facilities.

2 The Network Hardware

This work has been carried out on a Jetstream network [1]. Features of this network influenced some of the design decisions, so we'll briefly describe its relevant characteristics.

Jetstream is a Gbit/s token-ring network using copper coaxial cable for the physical link. The host interface for a HP Series 700 workstation consists of two cards forming a single entity (see Figure 1) – one card, Jetstream, connects to the network and provides the MAC layer; the other, Afterburner (see [2]), connects to a workstation bus and provides 1 MB of network buffer memory. Afterburner supports single-copy implementations of network protocols, allowing application data to be copied directly into these buffers without being copied into kernel buffers first.

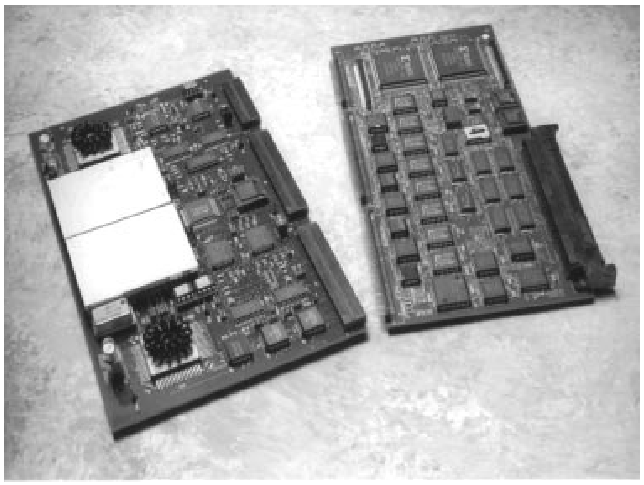
FIGURE 1. Jetstream & Afterburner

The Jetstream network uses the AAL-5 [3] frame format and provides node addressing and hardware demultiplexing of logical data streams via the use of ATM Virtual Circuit Identifiers (VCIs). A VCI consists of 15 bits – 4 bits indicating the Ring ID which identifies a particular node on the network, 10 bits identifying the logical data stream on that node, and 1 bit indicating whether this is a multicast or uni-

* Department of Engineering, University of Cambridge.

cast packet. Every packet sent on the network contains a VCI indicating the packet's destination.

3 Low-level access to the Jetstream driver

During the development of the Jetstream device driver we decided to provide a low-level access method to allow applications direct control of Jetstream facilities. The key goals of our scheme were to:

- Separate demultiplexing from protocol processing,
- Allow protocol code to be moved out of the kernel into user-space by providing interfaces that allow user-space protocols to execute efficiently

At the outset we decided that it was essential that certain facilities remained kernel-resident. For example, packet demultiplexing should remain in the kernel to avoid excessive context switching overhead. Also, packet buffering should remain in the kernel to minimise data-copying overhead. Since so many facilities hinge around the management of 'pools', or sets of packet buffers, we decided to call our access model the "Pool Model". Thus:

- A 'pool' is a set of fixed-size buffers uniquely associated with an application data stream
- A small set of operations can be performed on pools, such as allocating and freeing buffers, copying data in and out of buffers
- Arbitrary combinations of pool buffers may be formed into a packet and sent out 'on the wire'
- Applications provide the driver with sufficient information for it to place incoming packets into the appropriate pool
- Applications associate resource limits and policies with their pools allowing intelligent decisions to be made about packets as soon as they are received

Some of the pool facilities are assisted by features of the Jetstream/Afterburner hardware. For instance, associated with each pool is a Transmit VCI (TxVCI), at least one Receive VCI (RxVCI), and a number of Transmit (Tx) and Receive (Rx) buffers. Tx buffers are allocated dynamically from the Afterburner RAM up to a limit which is set on a per-pool basis (with a system-wide upper bound). Data is then copied into these Tx buffers and a sequence of buffers (up to a total of ~64KB) is then sent as a single Jetstream PDU. When a packet arrives at the interface, Rx buffers are allocated in the pool associated with the packet's destination VCI and the packet copied into them. If the pool already has all of its Rx buffers in use the packet is discarded.

These Tx and Rx operations (and all other operations) are communicated to the Jetstream device driver via ioctl() functions. Because of the high overhead associated with making such a system call, user-space processes can create 'scripts' consisting of a series of operations to be performed, then execute a whole script in a single ioctl(). This allows us to amortize the cost of crossing the user-kernel boundary across multiple operations. For further details of this model see [5].

4 Implementing TCP

We had already implemented a number of protocol libraries that made use of pools and these operation scripts, in particular an RPC library and a simple reliable byte-stream protocol library. Both of these exhibited performance at least as good as their kernel counterparts. The byte-stream protocol however, was too simple for 'real-world' use. For example, when closing a connection it blocked the process until all data had been acknowledged. Furthermore it functioned properly only when the process called on it regularly – the process could not be suspended. To show that our techniques had more general validity we decided to attempt the implementation of a user-space TCP.

4.1 Problems implementing TCP in user-space

There were three main problems in implementing TCP in user-space. The first problem was the difficulty of implementing TCP *per se*. Our interest was in making TCP work in user-space, not in developing TCP from scratch. The second problem was the asynchronous nature of protocol processing in TCP. Packets arrive and timers expire at arbitrary moments. How could user-space TCP handle these events without interfering with the application program? If asynchronous events are mapped to application level asynchronous processing (i.e. signal handlers), how could the application program's use of signals and timers be honoured? The third, and probably the most difficult problem was that TCP connections live longer than processes. For example, data passed to send() is delivered and the connection is closed even after the application has terminated. Also, the connection must survive even if the application has been suspended.

4.2 Design Goals

Our primary goals were to provide an implementation of TCP that was as complete and robust as possible and that achieved the highest performance possible. In a kernel TCP, the main way to increase throughput is by reducing data-touching. In a user-space TCP other techniques are also necessary such as minimising the number of system calls and context switches. While the pool model allows us to reduce the system call overhead, reducing context-switching is more difficult.

4.3 Design Overview

The first major decision taken in the design process was to base our implementation on an existing TCP/IP implementation rather than starting from scratch. The main factor motivating this decision was the difficulty associated with implementing a complex network protocol such as TCP. It was felt that it would be much easier to modify an existing protocol stack implementation, a decision heavily influenced by the availability of a HP-UX kernel (HP-UX 9.01, a derivative of 4.3BSD [6]) modified to take advantage of the extra features provided by the Jetstream network interface. This ‘single-copy’ kernel would be used as the starting point for our implementation and modified to operate in user-space. An additional benefit gained from this decision was that any differences in performance between our user-space implementation and the kernel implementation would be due mainly to the move to user-space, rather than any differences in the protocol code itself.

Once this decision had been made, we then had to consider the implications of moving a kernel implementation to user-space. TCP is a reliable, byte-stream transport protocol, i.e. it guarantees delivery of an ordered sequence of bytes over a network which may be unreliable and/or packet-oriented. The basic mechanism used to guarantee delivery is one of positive acknowledgement and timed retransmission of unacknowledged data. Performing this asynchronous TCP retransmit processing posed the first problem for our TCP implementation. We would need to use alarm timers and provide a handler for the alarm signal, but without preventing the application from also using alarms and signals. If the application was using alarms and signals then we would need to silently take control of alarm handling. If the application was not using these facilities we would need to put wrappers around system calls in order to restart them when they were interrupted.

The next problem we encountered was due to the semantics of the send() system call. Once TCP has indicated to an application that data has been sent, it must ensure that data has been received by the receiver, even if the sending application terminates. In a kernel implementation, the protocol state data is stored in the kernel’s data area so the kernel can continue to process the data even after the application has finished. We would need to provide some scheme whereby an application terminating would not cause all the protocol state and unacknowledged data to be lost. Furthermore we would have to accomplish this while still allowing the application to terminate.

Consideration of these problems led us to a design analogous to the partitioned design of the kernel. In 4.3BSD the kernel is partitioned into two halves – top and bottom. The top half consists of the system call interface used by applications, and the bottom half handles interrupts (including clock interrupts, i.e. events triggered by timers are handled by the bottom half). Both halves have access to the same data but do not communicate directly. A top-half function can block, e.g. while waiting for a resource to become available, suspending the process until either a bottom-half function or another process in a top-half function wakes it up. A process executing in the top half of the kernel cannot be pre-empted by another process but can be interrupted (causing a bottom-half function to be called). Because this bottom-half function may wish to access the same data as the top-half process, some form of concurrency control is needed. To accomplish this (on a uniprocessor) the kernel provides a mechanism whereby a function executing in a critical region of code can disable interrupts.

Our implementation closely follows this approach by using two processes, analogous to the two halves of the kernel. The application process (known as the user process hereafter) behaves like the top half, with the necessary system calls (e.g. socket, bind, accept) provided by a library of user-space functions. A second process (known as the child process hereafter) handles the bottom-half events e.g. receiving incoming data and processing timer-related events. Note that if our operating-system environment provided multiple threads per address-space, it would be possible to dedicate a thread in each user process to perform the bottom-half functions. However, we would still need at least one other process to handle reliably delivering data on termination.

Since both processes require access to protocol state, it must be stored in shared memory and protected by some form of concurrency control. In user-space this concurrency control is complicated by the fact that the bottom half (child process) can now be pre-empted by the top half (user process). While there is no way to prevent this possibility under normal HP-UX scheduling we can arrange that the top half yields if it detects that the bottom half has been pre-empted. Storing state in shared memory and having a separate child process allows us to handle reliable delivery in the face of application termination, and allows us to hide all asynchronous activities from the user’s program. While this approach satisfies our robustness goal it impacts performance as the system must continually context-switch between the user and child processes. We discuss this problem later.

4.4 Design Specifics

We now discuss in more detail the specific design choices we made regarding the partitioning of TCP functionality between the processes and describe the mechanisms we implemented for concurrency control, inter-process communication etc. We also describe some of the difficulties we encountered in trying to move kernel code into user-space with the minimum of changes.

4.4.1 Daemon Child

We decided to implement the child process as a child of a system-wide daemon rather than as a child of the user process. The main reason for this was to protect the child process from the user; if the child process was spawned by the daemon then it would not be able to be killed accidentally (or maliciously) by the user. This approach also allows us to give the child process special privileges (such as the ability to run at real-time priority). While our implementation uses a child process for every application using user-space TCP, it would also be possible to use the system-wide daemon itself for all bottom-half processing. In a non-prototype implementation the latter scheme might be preferable as it would consume fewer system resources, although it would introduce a single point of failure. In our implementation this system-wide daemon is also responsible for establishing TCP connections, so hereafter we refer to it as the connection server (see 4.4.7).

Creation of the child process occurs by the following mechanism. When an application uses the socket() function for the first time a message is passed to the connection server using our message passing scheme (described below). The connection server then forks the child process, which proceeds to create and initialise the shared state data. Once all the shared data has been initialised the child process sends a message back to the user process indicating its successful creation and passing back the identifier of the shared memory segment, which the user process then attaches to its own address space.

The main role of the child process is analogous to that of an interrupt service routine, i.e. to handle asynchronous events which occur. In a TCP/IP protocol stack these events fall into two categories – packet arrivals and timer ticks (see below). Timer ticks are indicated to the process by the raising of a signal, so they are inherently handled asynchronously. Unfortunately, for packet arrivals, the Jetstream device driver does not support the asynchronous I/O mechanism, so it is not possible to arrange for a signal to be raised when a packet is received. Instead, the child process must use the select() system call to wait for a packet to arrive on one of its pools. This leads to a very simple structure for the main loop of the child process:

- Wait on a select() system call for a packet to arrive.
- If a signal is raised then the select() call will be interrupted and the signal can be handled.
- Otherwise the select() call blocks until a packet arrives on one of our pools, from where the packet can be processed and passed up the protocol stack.

4.4.2 Shared Memory

As stated earlier all the protocol state data in our implementation is stored in shared memory. Unfortunately, in the HP-UX kernel TCP code, not all state data is allocated dynamically from the heap. For example, some kernel variables are in statically allocated memory. In order to make this data available to both processes it must be located in shared memory. To allow us to use as much of the kernel code as possible directly, we used a simple form of aliasing to map the kernel static variables onto variables in shared memory.

Another complication is that the shared protocol state contains process-specific information such as function addresses. As our child process is forked from the connection server and not from the user process, these addresses are different. Basically, all pointers to objects which are not in shared memory need to be different in each process. Thus every such pointer must be replaced by a pair of pointers, and a conditionally-compiled alias used to select between them.

Finally, since all of our protocol state has to be in shared memory, we were unable to use the normal malloc library as this allocates memory from the per-process heap. Thus we were forced to develop a shared memory allocator from scratch that faithfully emulated all the ‘features’ of the kernel allocator, such as returning specially aligned mbufs etc.

4.4.3 Concurrency Control

It is necessary to provide some form of control over access to the protocol state data in shared memory. The ideal approach is to use a fine-grained locking mechanism where each data structure has its own lock, and a process only locks data structures it is modifying. Although this allows for a high degree of concurrency between processes it is complex to implement.

We decided to use a very coarse-grained method of access control, where a process has exclusive access to the whole of shared memory while executing any of the protocol stack functions. This was the approach taken by the existing HP-UX kernel implementation, and so, again, required the minimum amount of change. A process must acquire a shared-memory lock before executing any protocol stack functions, and must not release the lock until it has finished executing those functions (unless the process is about to block). Furthermore, a process cannot pre-emptively take the lock from the other process.

HP-UX provides entities called semaphores which are ideally suited to coarse-grained access control. However, our desire to minimise the number of system calls led us away from this straightforward approach to try and find a mutual exclusion mechanism which could be implemented using variables in shared memory. The solution we initially decided to use was a version of Peterson's algorithm which we modified to yield rather than busywait (see A.1), although performance testing later led us to change this to a home-grown algorithm (see A.2).

4.4.4 Inter-process Communication

A mechanism must be provided for the user and child process to communicate with each other. We decided to use some form of message passing scheme, and considered several alternatives – UNIX domain sockets, pipes/named pipes (FIFOs), and shared memory in conjunction with some form of signalling mechanism e.g. signals or semaphores. It is important to consider the nature of the communication between the processes.

Whereas the user process can request a particular service from the child process at any time, the child process only ever sends a message to the user process when the user process is expecting one, either as a 'wakeup' message or as a reply to a request from the user process. Because the child process has to use select() to wait for packet arrival, we need our IPC mechanism to co-exist with this select() mechanism. We also intended to optimise the child process for the case where it was only receiving data on a single Jetstream pool by using a blocking RX operation rather than the select(). This effectively narrowed down our possibilities to those that raised a signal, as a signal can interrupt these operations whereas a semaphore wakeup cannot. We finally decided to use UNIX domain datagram sockets with the asynchronous I/O option to raise a signal when a message arrived at the child. The other factor motivating this decision was the fact that message passing maps directly onto a datagram-based protocol more naturally than a byte-stream protocol.

4.4.5 Binding of Addresses to Sockets

Our intention was for our user-space protocol stack to co-exist with a kernel protocol stack. One area where this presented a problem was in the binding of TCP/IP addresses (port + interface IP address pairs) to sockets. We had to make sure that there was no address binding conflict between the user-space and kernel protocol stacks, i.e. that two distinct sockets, one using a kernel protocol stack and one using a user-space stack did not both bind to the same address. To make sure this condition is always true we create a dummy kernel socket, bind the address to it, and if the bind succeeds also bind the address to the user-space socket. In this way we keep the system-wide TCP address space consistent.

4.4.6 Timers

The kernel provides network protocol stacks with a facility to request that the kernel calls a specified function after a certain time. The TCP/IP protocol stack uses this facility to implement two periodic timers to handle all time-dependent events. Our user-space implementation emulates this facility using the system real interval timer (note that we can not use select timeouts because in certain cases we optimise out the select() and just do a blocking receive). The child process keeps a list of all the timeouts requested, then sets the real interval timer to raise a signal when the shortest time-out expires. When the time-out expires, its length is subtracted from all other timeouts, it is removed from the timer list and the time-out function is called. For periodic timers the time-out function's actions always include re-installing itself in the list of timers.

4.4.7 Connection Server

We required a mechanism for assigning VCIs to TCP connections at connection time in a manner that did not require changes to application programs. We decided to use a 'TCP connection server'. This is a trusted process which runs on every network node that uses user-space TCP. It listens on a well-known VCI for connection set-up requests, and in our implementation it is the same process which forks the daemon children. When one application (the connector) wishes to connect to another application (the listener) at a TCP port on a remote node it goes through the following sequence of actions:

- The connector obtains an RxVCI for its pool from the Jetstream device driver. This is the VCI that will be used to receive Jetstream packets sent from the remote node.
- The connector sends a normal TCP SYN segment to the connection server on the remote node via the well-known TCP VCI.
- The Jetstream driver on the connecting node automatically inserts the connector's RxVCI in an unused AAL-5 trailer field of the SYN packet, then sends it to the remote node.
- When the connection server receives this SYN segment it extracts the RxVCI from the AAL-5 trailer then checks to see if there is a process listening on the port number specified in the TCP header (as in normal TCP connection establishment).
- If not, the segment is forwarded to kernel TCP so that the kernel can either pass the segment on to a process listening using kernel TCP, or in the absence of any kernel listeners send a RESET segment.

- If, however, the connection server finds that a process is listening on the correct port, it passes the SYN segment and the connector RxVCI on to the listener process.
- The listener receives the SYN, sets its TxVCI to be the RxVCI passed with the SYN packet, obtains a RxVCI for itself from the Jetstream driver, then sends the SYN+ACK reply back to the connector.
- The Jetstream driver on the listening node automatically inserts the listener's RxVCI in the AAL-5 trailer field.
- The connector receives the SYN+ACK and sets its TxVCI from the RxVCI passed in the SYN+ACK packet.

In this way, the two processes have now established a direct link without the user application being aware of VCIs in any form.

4.4.8 Kernel to User-space Communication

In order for our user-space protocol stack to communicate with a kernel protocol stack and vice-versa we use the facilities provided by the connection server. For convenience hereafter we refer to a process using the user-space protocol stack as a user-space process, and a process using the kernel protocol stack as a kernel process.

The kernel protocol stack normally has two VCIs associated with its pool – the IP VCI and the IP_DIVERT VCI. An externally visible kernel variable determines which VCI the kernel uses to send data (normally the IP VCI), while the kernel normally receives data on either VCI. Thus, all kernel TCP connections are multiplexed onto this single fixed VCI, so if a kernel process is to talk to a user-space process it needs the help of the connection server.

The connection server uses the IP_DIVERT VCI as the well-known TCP VCI, so on startup it must unbind the VCI from the kernel's pool and bind it to its own pool. Any Jetstream packets subsequently sent to the IP_DIVERT VCI on this host will then be received by the connection server rather than the kernel, allowing the connection server to decide whether to give the packet to a user-space receiver or divert it back to the kernel. Any process using the user-space protocol stack can ask the server to give it all packets on a particular connection by passing a special message to the server.

Although this scheme does permit heterogeneous communication there is a high overhead associated with using the connection server as an intermediary. This could be avoided either by making the kernel aware of VCIs at the link layer, or alternatively by making the Jetstream driver perform full TCP demultiplexing rather than simple VCI demultiplexing. Note that this need not compromise our goal of removing protocol code from kernel space as the protocol demultiplexing can be achieved using some generic filter specifications as in [7],[8] or [9].

4.5 Performance Optimisations

4.5.1 Buffer Caching

In the single-copy kernel, Jetstream buffers are allocated and freed individually, i.e. if 16k of buffer space is needed the allocate function is called 4 times (assuming 4k buffers) rather than just once. A naive user-space implementation simply maps the kernel functions to allocate and free buffers onto scripts which are executed to perform the same task. Because each script that is executed requires a system call to be made, this approach causes a large proportion of time to be spent in this system call overhead.

There are two possible solutions to this problem. The first is to batch operations together into a single script so that only one system call is necessary to execute a series of operations. The second approach is to use a cache of Jetstream buffers, i.e. a list of Jetstream buffers which are not currently in use on the associated pool. When a buffer is requested, one is returned from the cache if possible. If not, a number of buffers are allocated in a single script and added to the cache, then one returned to the caller. When a buffer which was allocated from the cache is freed it is returned to the cache. This approach has the advantage of being completely invisible to the protocol stack, but needs some kind of cache-flushing mechanism to prevent buffers being held in the cache for longer than necessary, since this makes them unavailable to other applications using the Jetstream interface.

4.5.2 Real-time Child Process

In HP-UX user processes may be designated as 'real-time' processes. A real-time process is given a priority from 0 (highest) to 127 (lowest), set by the application (or by another application with appropriate privileges). Real-time processes always have priority over all non-real-time processes, and cannot be pre-empted except by a higher priority (numerically lower value) real-time process or an interrupt. This behaviour is identical to that of kernel bottom-half functions, so it seems natural to make the child process a real-time process.

Making the child process real-time does in itself increase TCP performance but also introduces new problems. Normally, when a packet arrives on a pool, the child process (which is blocked on a select() call) is unblocked and made runnable, but doesn't run immediately. This gives the user process time, if it is executing protocol code, to complete its execution and release the shared memory lock before the child process is scheduled, runs and tries to acquire it. This causes the processes to become synchronised and hence the child process is normally able to acquire the lock immediately.

If the child process is real-time, when a packet arrives the child process will be unblocked from the select() call and instantly made runnable. If, at this point, the user process is already executing protocol code, the child will try to acquire the lock, fail, and be forced to yield. This introduces two unnecessary context switches. In our implementation this problem is exacerbated by the way we yield. We give up the processor by calling select() (with no descriptors set, purely as a 'sleep' function) with a timeout of 1ms. Unfortunately, the granularity of timer processing means that the child process sleeps on average for 5ms. This is much longer than the user process requires to finish executing its protocol code, so the processor idles for large amounts of time.

We tried to fix this yielding problem but it soon became clear that it was non-trivial to combine the 'no system calls in best case' advantage of Peterson's algorithm with acceptable synchronisation. We decided instead to opt for the simplicity of just using semaphores in the hope that what we gained from synchronisation would more than compensate for the extra system calls.

Although this seemed like a good solution, further consideration of the properties of real-time processes showed that the problem of combining 'no system calls' with synchronisation can be simplified by assuming that the user-process never runs at a higher real-time priority than the child process. In this case we know that the child process always executes until it blocks. In the context of our user-space TCP, this means that once the child has acquired the lock it cannot be pre-empted by the user process, i.e. it executes all of its protocol code then releases the lock before the user process can do anything. This leads to a simple mutual exclusion algorithm which requires no system calls in the best case and uses semaphores for synchronisation (see A.2).

Note that this solution is not applicable in a multiprocessor environment, nor does it address our original problem, the two unnecessary context-switches. In our conclusion we discuss some alternative techniques that would allow context-switches to be eliminated altogether in the common case.

4.6 Limitations

One limitation inherent in any user-space implementation of a network protocol stack is that it is very difficult to allow multiple processes to access the same socket, either as a child process inheriting open files across a fork() (and possibly exec()) system call, or by the rights-passing mechanism available with Unix-domain datagram sockets. While this is relatively easily implemented in the kernel it is extremely difficult to implement in user-space: multiple applications must all have access to the protocol state data in shared memory, so concurrency control is now across multiple applications and a single child process. Where we previously had two sets of function pointers in shared memory we must now have an arbitrary number of such pointers (assuming we wish to handle exec() calls and rights-passing – if not, all processes still use the same address space since they are all descendants of a common ancestor). One point worth noting is that even if several applications all access the same socket, all the asynchronous processing (packet reception, timer events) will still be handled by a single child process, i.e. that associated with the process which created the socket.

4.7 Security considerations

During the course of this work we have ignored the issue of security as in our scheme each TCP conversation is associated with a unique link-level channel or VCI. Thus, a malicious application can only affect packets travelling on its own dedicated VCI, which limits the scope of the damage it can inflict to just the peer application. In cases where our user-space TCP communicates with a kernel TCP, the connection server, a trusted component, acts as an intermediary and could ensure that all TCP packets were well-formed. More generally if an application does have direct access to networking then mechanisms have to be in place to check the validity of packets transmitted. One approach (as suggested in [10]) is for a trusted entity to install some sort of template in the transmit path, which packets must successfully match before being transmitted.

5 Performance

In this section we present application-to-application throughput measured over Jetstream using various TCP configurations. As the receiver is always the bottleneck in our demonstrations (i.e. 100% busy) we also present the send-side CPU utilisation for the different TCPs.

The measurements reported here were collected between two HP 9000/735 workstations running HP-UX 9.01 and using the **netperf** [11] utility. Both workstations were connected to the site Ethernet and had the usual background processes running. In all cases TCP window scaling [12] was used with socket buffers of 245760 bytes. The Jetstream PDU size was 61504 bytes.

5.1 Throughput of user-space TCP

Figure 2 shows the measured throughput in Mbit/s as we increase the application message size. One configuration uses the single-copy kernel TCP, another configuration uses a normal dual-copy TCP and the other configurations are all various versions of our user-space TCP.

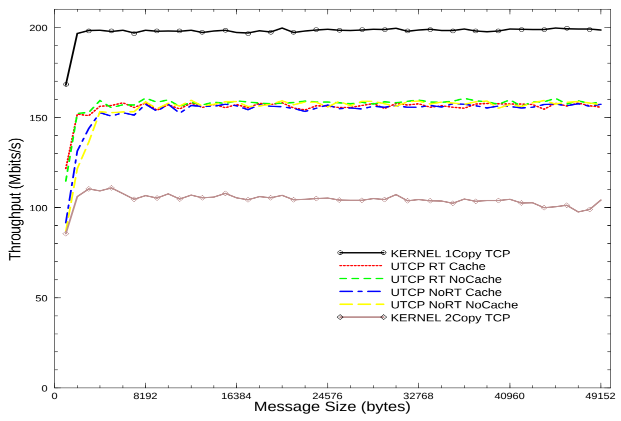

FIGURE 2. TCP throughput

It is very clear that reducing the number of copies from two to one has a dramatic effect on the throughput. It is also clear that our user-space TCP outperforms the normal kernel implementation and even achieves 80% of the performance of our single-copy kernel implementation. The various flavours of user-space TCP all exhibit the same performance, approximately 160 MBit/s. The buffer cache optimisation mostly benefits the TCP sender, so this is not remarkable. However, the real-time priority optimisation should benefit both sender and receiver, so the fact it has no impact on throughput is curious. We conjecture that the benefit of reducing system-calls is being counteracted by the additional unnecessary context-switches described in 4.5.2.

5.2 CPU utilisation of user-space TCP

Figure 2 shows the measured send-side CPU utilisation as we increase the application message size for the same TCP.

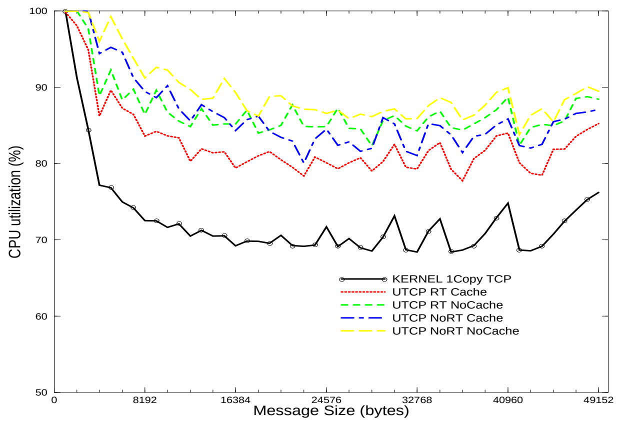

FIGURE 3. TCP send-side CPU%

Here, the effect of our optimisations is clear. The user-space TCP implementation with both optimisations uses the least CPU, about 82%; the implementation with neither optimisation uses the most CPU, about 88%. In all cases, kernel TCP uses substantially less CPU than our user-space versions, about 72%, which is what we would expect. If our experiments were sender-limited rather than receiver-limited the effect of the optimisations would be visible on throughput.

6 Related Work

Our work has a great deal in common with previous user-space TCP implementations. Both [10] and [13] describe schemes for the Mach [14] operating system that allow a complete TCP to be provided using a user-linkable library and a trusted connection server. In these implementations, as in ours, the common TCP send/receive path is provided by threads of control executing in the linkable library. The key differences of our scheme are that firstly, our work exploits some features of an underlying high-speed network interface that allow a genuine ‘single-copy’ TCP to run in user-space. Secondly, our techniques allow us to achieve TCP throughput at least an order of magnitude greater than previously reported results. Thirdly, our TCP implementation is for a Unix-like operating system and thus achieves high performance despite being unable to make use of desirable features such as multiple threads per address-space.

7 Conclusions and Future Work

We have implemented a TCP in user-space which performs at 80% of our single-copy kernel implementation. We believe this shows that protocols can be developed and run in user-space at acceptable performance levels, and we conclude that user-space protocol implementations need not perform poorly if they are able to exploit an appropriate set of low-level interfaces.

One of the key issues we wished to investigate in this work was whether with the appropriate partitioning a complex network protocol could be moved into user-space. It is clear that low-level demultiplexing and buffer management are functions that really need to stay in kernel-space. Ideally, demultiplexing should be on upper-layer protocol headers; however, channel or VCI demultiplexing is an acceptable compromise. The ‘pool model’ we describe seems to provide a perfectly acceptable interface between user-space and kernel-space functions allowing applications to exert very fine control over their data-streams. Also, batching pool operations into scripts is a simple and effective way of amortizing the cost of system calls.

During the course of developing our user-space implementation it became clear that implementing a full-blown transport protocol like TCP was a decidedly non-trivial task. Our goals were to provide a complete, robust implementation that achieved high-performance. Unfortunately, some of these goals are mutually incompatible. To provide a complete, robust implementation we were forced to adopt a multi-process architecture; as a consequence of this choice our TCP was constantly context-switching, which lowered performance. Also, the multi-process nature of the implementation meant we seemed to spend as much time worrying about concurrency control as about our real goals.

In future work, we hope to investigate techniques for providing a complete TCP implementation in a single-process without compromising robustness. One possible approach is to exploit the fact that most application programs use sockets in 'blocking mode', sleeping on the socket till there is more data to receive or space available for sending. If a packet arrives during this 'sleep' we could perform receive processing in the context of the application process, which would eliminate context-switches in the common case. With these improvements even better performance should be achievable.

References

[1] G. Watson, D. Banks, C. Calamvokis, C. Dalton, A. Edwards and J. Lumley, 'AAL5 at a Gigabit for a Kilobuck,' *Journal of High Speed Networks,* Vol. 3, No. 2, pp. 127-145, 1994.

[2] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards and J. Lumley, 'Afterburner,' *IEEE Network Magazine*, Vol. 7, No. 4, pp. 36-43, July 1993.

[3] CCITT, 'AAL Type 5, Draft Recommendation text for section 6 of I.363'. CCITT Study Group XVIII/8-5, Report of Rapporteur's Meeting on AAL type 5, Annex 2, Copenhagen, 19-21 October, 1992.

[4] D.L. Tennenhouse, 'Layered multiplexing considered harmful,' In *Proceedings of the 1st International Workshop on High-Speed Networks*, pp. 143-148, November 1989

[5] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis and C. Dalton, 'User-space protocols deliver high performance to applications on a low-cost Gb/s LAN,' in *Proceedings SIGCOMM 1994*, London, September 1994

[6] S.J. Leffler, M. McKusick, M. Karels and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[7] J. C. Mogul, R.F. Rashid and M.J. Accetta. 'The Packet Filter: An efficient mechanism for user-level network code,' in *Proceedings of the 11th ACM Symposium on Operating System Principles*, pp. 39-51, November 1987.

[8] S. McCanne and V. Jacobson, 'The BSD Packet Filter: A New Architecture for User-level Packet Capture,' in *Proceedings of the 1993 Winter USENIX Conference*, pp. 259-269, January 1993.

[9] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson and P. Sarkar, 'PathFinder: A Pattern-Based Packet Classifier,' in *Proceedings of the 1st Symposium on Operating System Design and Implementation*, November 1994.

[10] C.A. Thekkath, T.D. Nguyen, E.Moy and E.D. Lazowska, 'Implementing Network Protocols at User Level,' in *Proceedings SIGCOMM 1993*, San Francisco, pp. 64-73, September, 1993.

[11] Rick A. Jones, '**netperf**: A Network Performance Benchmark,' Revision 1. Information Networks Division, Hewlett-Packard Co., March 1993. The **netperf** utility can be obtained via anonymous ftp from hpux.csc.liv.ac.uk in /hpux/Networking/Admin.

[12] V. Jacobson, R. Braden, D. Borman, RFC 1323, 'TCP Extensions for High Performance'. May 1992.

[13] C. Maeda and B.N. Bershad, 'Protocol service decomposition for high-performance networking,' in *14th ACM Symposium on Operating Systems Principles*, December 1993.

[14] M.J. Accetta, R.V. Baron, W. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian Jr., and M.W. Young, 'Mach: A New Kernel Foundation for Unix Development,' in *Proceedings of the 1986 Summer USENIX Conference*, pp. 93-113, July 1986.

[15] G.L. Peterson, "Myths about the mutual exclusion problem,' in *Information Processing Letters*, Vol. 12, No. 3, pp. 115-116, June 1981.

A Appendices

A.1 Peterson's algorithm

Peterson's algorithm for mutual exclusion between two processes is as follows (also see [15]):

```
Lock()
{
    wantAccess[ME]=TRUE;
    nextAccess=YOU;
    while (wantAccess[YOU] && nextAccess==YOU)
        /* Add yield code here */
        ;
}
Release()
{
    wantAccess[ME]=0;
}
```

A.2 Edwards' Algorithm

This algorithm for mutual exclusion exploits the fact that the child process runs at real-time priority and so cannot be pre-empted by the user process:

```
Lock()
{
#ifdef CHILD_PROCESS
    if (disable_child) {
        /* If user process has disabled us
         * then we block and wait for him to
         * wake us up
         */
        child_blocking=TRUE;
        WaitOnSemaphoreForWakeup();
    }
#else
    /* We know child must be blocked at the
     * moment (thus not holding lock) as it
     * runs as a real-time priority so can
     * not be pre-empted
     */
    disable_child=TRUE;
#endif
}
```

```
Release()
{
#ifdef CHILD_PROCESS
    if (wakeup_user_process) {
        WakeupSleepingUserProcess();
    }
#else
    disable_child=FALSE;
    if (child_blocking) {
        child_blocking=FALSE;
        SignalWaitSemaphore();
    }
#endif
}
```

A.3 Control Messages

Our message passing scheme uses a total of 16 different messages passed between the 3 processes used in the implementation (user, child, and connection server). These messages are described below.

- **Listen** (child to server) – sent as a result of the child receiving a 'Listen_Child' message, it notifies the connection server which child process is listening on which TCP port (and also when a user closes a listening socket). This association between ports and child processes allows the server to send a 'Syn' message to the correct child process.
- **Forward** (child to server) – serves a similar purpose but tells the server to forward any packets arriving on a specified connection (identified by both end-points) to a specified VCI i.e. to the appropriate child process.
- **Divert** (child to server) – response to a 'Syn' message once a pool has been created to divert the SYN packet into. If, for some reason, the connection is not completed and a new socket is not created this pool will be closed at the appropriate time.
- **Syn** (server to child) – sent by server on receipt of a packet with the SYN flag set and a destination port which is being listened to by a child process.
- **Spawn** (user to server) – initial message sent from user process to connection server telling server to spawn a child process. Once the child has been forked and initialised, it (the child) sends an 'Init_Reply' back to the user process.
- **New_Socket** (user to child) – sent to the child when a socket is created, its purpose is to allow the child process to complete the initialisation of the new socket data structure e.g. set the values of its function pointers, create a pool for the socket. The child sends back a 'General_Reply' indicating success or failure.
- **Bind** (user to child) – when a user process binds a socket to an address it sends the parameters of the call to the child process requesting the child to attempt the bind on the kernel bind-point (see 4.4.5). The child replies with a 'Bind_Reply' indicating the result.
- **Listen_Child** (user to child) – simply passes the parameters of the listen call from user process to child, causing the child process to send a 'Listen' to the connection server.
- **Connect** (user to child) – notifies the child process that the initial SYN has been sent and causes user process to block until it receives a 'Connect_Reply' indicating success or failure of connection attempt.
- **Set_Tx_Bufs/Set_Rx_Bufs** (user to child) – because the limit on number of buffers associated with a pool can only be changed by the process which originally opened a pool, the user process must hand these requests (from a setsockopt() call) off to the child process since the child always creates the pools associated with sockets.
- **Set_Reuse_Addr** (user to child) – this socket option must be applied to the kernel bind-point, so it is passed to the child process as the process which opened the bind-point.
- **Init_Reply** (child to user) – sent from child process to user process once child has completed initialisation, it contains the shared memory identifier to allow the user process to access shared memory.
- **Bind_Reply/Connect_Reply/General_Reply** (child to user) – these messages simply pass the results of a request sent to the child back to the user process.