User Space Protocols Deliver High Performance to Applications on a Low Cost Gb/s LAN

Aled Edwards, Greg Watson, John Lumley,
David Banks, Costas Calamvokis, Chris Dalton
Networks and Communications Laboratory
HP Laboratories Bristol
HPL-94-59
June, 1994

Gbit/s LAN,
jetstream, ATM,
TCP/IP, AAL5

We address two issues in high-speed networking: how to provide Gbit/s networking at low cost and, how to provide a flexible low-level network interface so that applications can control their data from the instant that it arrives. We describe the Jetstream Gbit/s LAN, an experimental, low cost network interface. Jetstream frames contain a channel identifier so that the network driver can immediately associate an incoming frame with its application. Applications can then control how their data should be managed without the need to first move the data into the application's address space. Measured results show that both kernel- and user-space protocols can achieve very good throughputs in excess of 200-Mbit/s.

Internal Accession Date Only

User-space protocols deliver high performance to applications on a low-cost Gb/s LAN

Aled Edwards, Greg Watson, John Lumley, David Banks,
Costas Calamvokis, and Chris Dalton

Hewlett-Packard Laboratories, Filton Rd, Bristol, UK

Abstract

Two important questions in high-speed networking are firstly, how to provide Gbit/s networking at low cost and secondly, how to provide a flexible low-level network interface so that applications can control their data from the instant it arrives.

We describe some work that addresses both of these questions. The Jetstream Gbit/s LAN is an experimental, low-cost network interface that provides the services required by delay-sensitive traffic as well as meeting the performance needs of current applications. Jetstream is a combination of traditional shared-medium LAN technology and more recent ATM cell- and switch-based technology.

Jetstream frames contain a channel identifier so that the network driver can immediately associate an incoming frame with its application. We have developed such a driver that enables applications to control how their data should be managed without the need to first move the data into the application's address space. Consequently, applications can elect to read just a part of a frame and then instruct the driver to move the remainder directly to its destination. Individual channels can elect to receive frames that have failed their CRC, while applications can specify frame-drop policies on a per-channel basis.

Measured results show that both kernel- and user-space protocols can achieve very good throughput: applications using both TCP and our own reliable byte-stream protocol have demonstrated throughputs in excess of 200 Mbit/s. The benefits of running protocols in user-space are well known - the drawback has often been a severe penalty in the performance achieved. In this paper we show that it is possible to have the best of both worlds.

1 Introduction

Networks will soon have to support new applications such as videoconferencing, shared authoring and remote training which will need synchronisation of streams of audio, video and graphical information. These new applications will cause two important changes in networked systems. First, the networks themselves must offer new services that provide guaranteed bandwidth and bounded access delay. Second, the way in which computers process networked information will change: the traditional model of just one or two kernel-based protocols handling all network traffic will be too inflexible to meet the wide range of new requirements.

Already we can see the first change, with recent network technologies such as FDDI and ATM providing a range of different services. The critical obstacle to their success is cost: LANs are commodity items, and for even a Gbit/s LAN to achieve substantial commercial success, it must be 'cheap'.

The second change is still to happen and is about computers being able to deal with different network data streams in many different ways, but without sacrificing performance. Experience with current networking shows that it is essential that data is not copied unnecessarily if both user and kernel processes are to achieve high rates of throughput. For example, consider a user process that is managing an incoming video stream where the start of each video frame identifies the position of the video data on the screen. When a frame arrives, the consumer process must examine the first few bytes of information, but then need only instruct the buffer manager to move the remainder of the data to a particular display location. Such piecemeal processing of network data will require much more flexibility in both kernel and user software and more intimate contact between the application and the network buffers than currently available: the model provided by normal TCP and UDP processing would force the destination process to consume all the data before anything further could be done.

In this paper we report on some work that addresses both of these issues. Our objective was to develop a network subsystem that provides a variety of services as well as high bandwidth and then to add a simple driver-level interface with the flexibility to support both kernel and user-space protocol processing. Two extra goals were that the network should be low cost and designed for the workstations we use day-to-day. These goals were entirely selfish: we want to be able to use a Gbit/s network as our own LAN and distribute it to other researchers so we will benefit from their experience.

We have developed such a low-cost, high-performance network that enables user applications to achieve throughput in excess of 200 Mbit/s, with an aggregate bandwidth of 800 Mbit/s. In section 2 we describe our experimental LAN, called Jetstream, and explain the decisions we made regarding the topology, frame size and core services. In section 3 we outline the software driver which provides flexible services right up to the user's application. In section 4 we describe how both the current kernel-based protocols as well as user libraries can exploit this driver. Section 5 presents the measured performance of the system.

2 The Jetstream LAN

This section describes the main characteristics of the Jetstream LAN and explains the reasoning behind the most important decisions. More detail is presented in [1].

2.1 Ring Topology

A Jetstream LAN interconnects up to 64 computers in a ring topology to provide a shared bandwidth of 800 Mbit/s. It is designed to interconnect a workgroup consisting of typically 10 to 30 computers and rarely more than 50. By focusing on this group of users we avoid trying to build a network that attempts to cater for all but risks satisfying none. We also restrict the 'circumference' of the LAN to a few kilometres.

We chose a shared-medium ring topology rather than a switch-based mesh because it costs much less. With a ring there is no expensive switch to be accounted for and only a single transceiver and cable is needed for each station. A switch-based network would offer greater aggregate bandwidth, but we think that 800 Mbit/s will be sufficient for many workgroups even allowing for uncompressed video streams. Current applications, such as networked file systems, exhibit very bursty behaviour which can be exploited by a shared medium LAN, with applications often finding an idle network and thus achieving immediate access. Such applications will remain a major source of network traffic for a long time to come.

2.2 Frames vs. cells

Jetstream uses variable-size frames up to 64 Kbytes in length, not small fixed-size cells. Segmentation and reassembly of small cells at very high speed is costly: either expensive special hardware is needed or the host processor has to perform the necessary operation and with current machines the resulting performance penalties would be unacceptable. If the network can provide the necessary services using variable size frames then there is no incentive to use cells.

An additional benefit is that frames are the unit of retransmission for many widely used protocols. Whereas a congested router will discard an entire frame, a congested ATM switch may discard just a single cell from a frame and propagate the remaining (useless) cells. In both cases the entire frame will have to be retransmitted so by dropping partial retransmission units the switch wastes bandwidth and may make congestion even worse.

We chose a maximum frame size of 64 Kbytes. Large frames are more efficient in terms of the useful (application) bytes sent per frame and reduce the total amount of processing needed to send and receive the application data. On the other hand, delay-sensitive traffic may be denied access to the network during the transmission of a large frame.

Two further reasons led us to choose 64 Kbytes as the maximum frame size. First, it is important that Jetstream can handle ATM AAL5 frames without the need for segmentation and reassembly, AAL5 being the ATM Forum's adaptation layer of choice for data and thus likely to be an important frame format in the future [2]. The second reason is that we did not know how end-to-end efficiency would be affected by frame size and so we wanted to experiment with a variety of sizes.

2.3 Frame format

The Jetstream frame format (Figure 1) is derived from the ATM cell format as defined by the ITU-TSS, but with two differences: the frame size can vary in multiples of four bytes from 56 bytes to 64 Kbytes, and there is an additional Jetstream header of three bytes which comes before the ATM header.

We chose to use the ATM format rather than that of a traditional LAN as the 24-bit ATM virtual channel identifier (VCI) provides sufficiently fine granularity that each logical data stream within a host can be multiplexed and demultiplexed at a single point: the driver. This is exactly what is necessary to address the second issue described in the introduction. By demultiplexing at the driver an incoming video stream could be processed differently to a file transfer stream or even another video stream using a different cod-

ing. Tennenhouse [3] provides other reasons why it is best to demultiplex only at a single point.

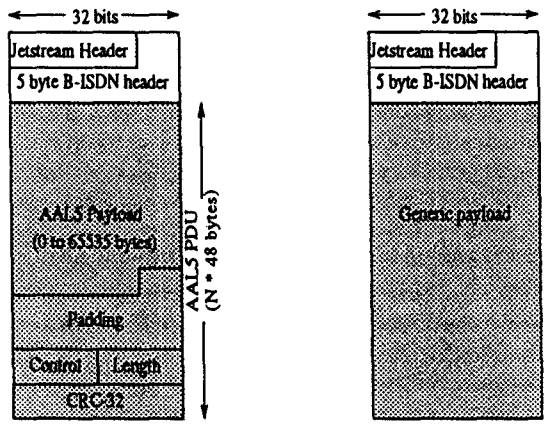

FIGURE 1. The Jetstream frame format

The three-byte Jetstream header serves two purposes: it aligns the payload to a 32-bit boundary, making data transfers more efficient for most 32-bit computers and it contains information which is useful in the operation of a ring-based LAN. This information is used to assign a unique identifier to each station when the network starts up. This identifier enables each station to use a unique part of the VCI space, which is shared between many stations on Jetstream's shared medium.

Using the ATM format means that it will be almost trivial to bridge a Jetstream LAN to an ATM network. A 53 byte ATM cell becomes a Jetstream frame with just the addition of the Jetstream header. An ATM/Jetstream interface could also reassemble ATM cells into Jetstream frames because, although this operation is expensive, it need only be done at one point in the whole network. The Jetstream header must be removed from frames passing to an ATM link and the payload segmented into 48 byte cells, each prefixed with the provided ATM header. Of course the virtual connection identifier may need to be changed at this interface.

2.4 Services

The network must support at least two classes of service: an asynchronous service and a service providing guaranteed bandwidth with bounded delay. To what degree should bandwidth be guaranteed, and to what timescale should access delay be bounded? There is clearly a spectrum of possible services, two examples being the multi-millisecond delay of the FDDI synchronous service and the once-per-125-microseconds isochronous service proposed in FDDI-II.

The Jetstream LAN uses a variation of the timed token protocol used in both IEEE 802.5 and FDDI. This was chosen because it is very simple (and cheap) to implement. Whereas FDDI provides eight priority levels for asynchronous traffic as well as a synchronous service, Jetstream provides a single priority asynchronous service and the synchronous service. Only one source is transmitting data at any one time and frames are removed by their source. A bit in each frame is used to detect frames that circulate the ring more than once.

A concern with using the timed token protocol [4] is the worst case access delay that synchronous traffic might encounter. This protocol uses a target token rotation time (TTRT) to bound the delay encountered by any station. The maximum delay encountered by synchronous traffic will not exceed twice the TTRT. Given a maximum frame size of 64 Kbytes it is quite feasible to define the TTRT to be one millisecond so that the maximum delay will be two milliseconds. It is not clear whether a two millisecond worst case access delay is acceptable or not, but if humans can tolerate delays of the order of 100 milliseconds [5] for audio/visual information then Jetstream can provide the services for many new applications.

2.5 The prototype LAN

We have designed and built a small number of Jetstream interfaces for use in our HP Series 700 workstations. Coaxial cable has been used as the physical layer interconnect for distances up to about 50 meters, driven by HP's HDMP-1000 serial transceiver [6] chipset, which employ a 16B/20B coding scheme and can drive the cable directly at rates beyond 1 Gbit/s.

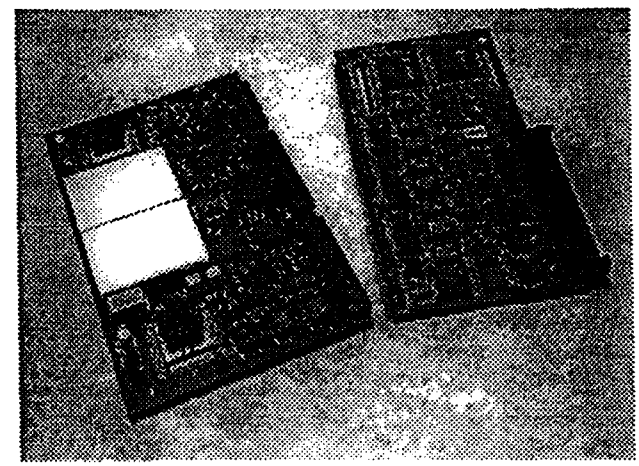

FIGURE 2. The prototype Jetstream interface (left) with Afterburner card

The Jetstream interface has been designed for use with the Afterburner network card [7]. Figure 2 shows both cards, which connect to form a single board that plugs into the graphics bus of the workstation.

Throughout this work we have aimed to produce a network interface that offers high performance at low cost and modest size. The current Jetstream prototype is built entirely from commercially available parts and costs less than $2000 at one-off prices. The logic for the timed token protocol, host interface and CRC and IP checksum calculations are contained within two field programmable gate arrays. Most of the other devices are needed to convert between ECL logic used in the transceiver and TTL used by the rest of the system (the two large 'boxes' on the left hand side of the card are simply voltage converters). A product could be made simpler and cheaper. The design uses a minimal amount of memory, so we could compress much of the random logic into a single ASIC and reduce the total cost to approximately $500.

Other comparable network interfaces include Orbit [8], the iWarp/Nectar CAB [9] and Davie's 622 Mbit/s ATM interface [10]. At 20cm by 12cm we believe that Jetstream is substantially smaller and cheaper than these.

2.6 Afterburner

The Afterburner [7] card is simply some multi-ported buffer memory together with some checksum calculation logic. It is designed to be used with a network card such as Jetstream. It is based on the ideas in Jacobson's WITLESS proposal [11] and our experience with the Medusa [12] FDDI card. With the WITLESS model, data is only copied once and the IP checksum is calculated during that one copy operation. Clark [13] and Partridge [14] describe these issues in much more detail.

Afterburner provides one Mbyte of buffer memory as well as IP checksum support for outbound data. The buffer memory can be organised as 512 2-Kbyte blocks, 256 4-Kbyte blocks, etc. A single frame can span several Afterburner blocks, so IP can use a maximum protocol data unit size of 64 Kbytes regardless of the actual block size. The choice of block size will affect the system performance: larger blocks yield better throughput (see section 5) but exhaust the available blocks more rapidly if many small frames are transmitted.

3 Making the interface appear fast, flexible and friendly to the user

Our goal in developing the Jetstream driver software was to support normal kernel-resident protocol stacks like TCP/IP and UDP/IP and also to support applications that require a greater degree of control over network data streams than the conventional send/receive model provides.

An example of such an application is one that routes video data from a network to a frame buffer. This application has no need to consume the data - it just needs to examine a portion of the data in order to be able to route it to the appropriate location in the frame buffer.

The same application, because of the nature of its data stream, might be willing to receive data that arrives corrupted, and if its receive buffers overflow might prefer old packets to be discarded rather than more recent ones. Neither of these options is available under the conventional send/receive model.

To provide support for this sort of application and to provide support for the development of efficient user-space protocols, we have developed a low-level access scheme that allows a much finer degree of control to be exerted by applications over their data-streams. In the case of Jetstream, the functionality is provided by the driver and exploits features of the hardware; however, we believe the scheme is more generic and can be made hardware-independent.

The key characteristics of our low-level access scheme are as follows:

- Application data-streams are associated with sets of fixed size kernel buffers - we call this set of buffers a 'pool'.
- Applications can allocate buffers to their pool (subject to an upper limit) and free buffers from their pool.
- Applications issue explicit operations to control movement of data between their kernel buffers and user-space.
- Applications can compose arbitrary combinations of buffers into packets for transmission on the Jetstream network.
- Applications provide the driver with sufficient information for it to place incoming packets into the appropriate pool.
- Applications may associate other characteristics with their pools, such as whether to keep or drop erroneous packets and whether to discard new packets in preference to old.

If the application is to provide enough information to allow the driver to demultiplex packets to the appropriate pool the driver either needs to have detailed knowledge of all possible higher layer protocols, or needs to understand some general filter specification [15].

Pool-specific policies for error handling are useful because particular data streams may have different requirements in case of error [16] or overflow. For instance, a file transfer application will drop PDUs with an incorrect link CRC and drop newly-arrived PDUs in preference to ones already received. A video application might accept PDUs containing errors, and drop old PDUs in preference to new ones, in order to keep the display up-to-date.

A number of other approaches allow a similar sort of low-level access. Traw [17] and Druschel [18] are excellent examples. These schemes perform demultiplexing in hardware whereas our scheme does it in software.

A more fundamental difference is that in these schemes data is available for consumption by an application immediately after it has arrived off the network. In our scheme the application must issue an explicit copy operation before the data is accessible. The main reason for this is that the copy operation allows us to provide a uniform interface to what may be a wide variety of underlying mechanisms; for instance, the pool buffers might reside in I/O space (on the network adapter) so actually copying the data might be necessary to save the application having to understand the peculiarities of accessing I/O memory; alternatively, the buffers might reside in system memory in which case the copy operation can (if the VM system is sufficiently clever) just swap the appropriate pages in the virtual memory maps and thus avoid actual data-copying. Another more prosaic advantage is that an explicit operation provides a convenient way of returning the checksum of the received data.

3.1 Pools: Facilities and Operations

A certain subset of the requirements we have identified are assisted by facilities of the Jetstream hardware. For instance, buffering is provided by Afterburner's on-board Mbyte of VRAM, and pools are merely dynamically allocated subsets of these VRAM blocks. Packets are demultiplexed to pools based on VCI, with the assistance of the Jetstream VCI lookup table: if no pool identifier is specified for a particular VCI then packets on that VCI are not received; what's more the Jetstream adapter does not even generate an interrupt for the packet. Thus, packets are only received if they are destined for local pools. The remaining requirements are satisfied entirely in software through the Jetstream driver's support for the 'pool' abstraction. The primitive operations supported on these pools are summarised in Table 1.

open	create a pool with a specified Tx limit, Rx limit and drop/copy policies
close	close the pool and free all associated blocks
alloc	allocate a block on the pool for transmission
free	free an allocated or received block
copy	copy data to/from a block owned by the pool and return a checksum
send	transmit a set of blocks using pool specific framing
assoc	associate a VCI with the pool
deassoc	break VCI to pool association
enqueue	enqueue received PDU (a set of blocks) on pool receive queue
dequeue	dequeue received PDU from pool receive queue

TABLE 1. Operations on pools

When a pool is opened its characteristics are described to the Jetstream driver by specifying a number of parameters. These include:

- the maximum number of blocks that may be used for transmission and reception.
- a receive function which handles incoming PDUs for this pool (and also determines the error-handling policy, the overflow policy and the link framing used on the pool).
- a data-copying policy and a data-copying result.

Specifying a limit on the number of blocks usable by each pool allows the Jetstream driver to distribute the Afterburner buffer space in whatever manner clients require. It also prevents clients from deliberately or accidentally using up all the available buffering.

The receive function is one of a small set of driver-supplied routines. Certain receive functions are dedicated to handling PDUs on a specific pool. Other receive functions are more generic and handle PDUs on a range of pools. Of the generic set, two append PDUs to a pool receive queue, but implement different policies on finding that the pool is full. One drops old PDUs; the other drops new PDUs.

The data-copying policy allows the pool client to determine whether data should be copied via the cache or not. For some clients having the data in cache is useful; for others it is actually a hindrance. The data-copying result allows the client to select the checksum operation that should be car-

ried out during the copy (currently only IP and NONE are supported).

4 Using the pool model

4.1 The Kernel as a Client

All accesses to Jetstream go through pools - it is the *only* access path. For instance, three pools are automatically created by the Jetstream driver for its own use. The INIT pool is used exclusively by the Jetstream driver for ring initialisation and station management. The other pools are used for ARP and IP traffic; PDUs received on either of these are passed to the corresponding protocol. To do this the Jetstream driver incorporates a 'glue' layer that uses the normal kernel interfaces and, in turn, provides the interfaces that the kernel expects.

The kernel sees what is apparently an Ethernet interface with a 64 Kbyte maximum frame size. Chains of memory buffers ('**mbufs**') containing Ethernet frames are passed down by the kernel in the normal way. The glue layer examines the frame header to determine whether to allocate blocks in the IP or ARP pool and generates the Jetstream header. It then copies the data portion of the frame into the blocks it has allocated and finally transmits the frame.

The driver receives inbound frames as a set of Afterburner blocks. The interrupt handling code uses the channel identifier to determine the correct pool, then the driver calls the function that is associated with the pool. For IP and ARP this function does the following: checks the CRC, strips the framing information, copies the frame to mbufs, puts a fake Ethernet header on the front and then passes the resulting mbuf chain up the protocol stack via the normal kernel mechanisms.

Currently, there are two VCIs for IP: one for broadcast and one for unicast. All IP traffic uses one or the other. At an end station the IP pool is configured to receive on the both VCIs. Thus, within a single host all IP traffic uses the same pool, so low-level demultiplexing is not exploited.

Despite this, adequate performance can be achieved, as shown in section 5.

4.2 Single-copy kernel client

In section 4.1, the pools were known only to the driver. If higher layers are also aware of pools, then PDUs can be demultiplexed directly to the end-consumer [3]. For instance, when a TCP connection is established, dedicated pools and VCIs could be allocated allowing incoming PDUs to be directly demultiplexed to the appropriate TCP control block and socket buffer. This of course would require changes to TCP code in the kernel. Whilst we believe that this is the correct approach, it does entail substantial modifications to protocol code, so instead, as an intermediate step, we extended some earlier work [7] to accommodate the pool model. Thus, we retain a single IP pool but make some of the IP pool operations visible to socket layer code. In this way we support what we call 'single-copy' operation [7][8] for TCP and UDP.

We made the IP pool **alloc**, **copy** and **free** operations visible to the socket layer by extending the table of function pointers within the **ifnet** structure. This allows the socket **send()** and **recv()** functions to copy data directly in to and out of IP pool blocks. A special mbuf type had to be used to differentiate normal data from that in the IP pool. Furthermore, TCP and UDP protocol code had to be modified slightly to handle these special mbufs appropriately.

Using this single-copy approach and the checksum calculating facilities available within Afterburner, excess data-touching operations can be eliminated and excellent performance results obtained (see section 5).

4.3 Other Clients

Whilst we have not exploited all of Jetstream's facilities from kernel code, we have seen no reason to restrict the scope of other Jetstream clients. Our express goal has been to make the full range of Jetstream facilities available to ordinary user processes. We see user processes as first-class clients of Jetstream and expect to develop new applications that exploit Jetstream's facilities to the full.

Currently in HP-UX, the interfaces between user processes and device driver code are not particularly rich, so we have been forced to go through some contortions to make Jetstream facilities available to user clients. A user client must first open the Jetstream device file **/dev/jetstream**, and then associate a pool with the returned file descriptor by issuing a special **ioctl()** call. Once the pool has been opened all other pool operations may be invoked by issuing other **ioctl()** calls.

Thus, all normal pool operations, such as **alloc**, **copy**, **free** and **send** may be used in whatever manner and in whatever order the user desires. To make receive operations possible, the user pool operations are augmented by two pseudo-ops: **wait_rx** causes the user process to sleep for a specified period or until a PDU is available to be received; **rx** does the (non-blocking) receiving.

This technique allows user processes to access all Jetstream services, but initial efforts showed us that the system call overhead involved in making all these **ioctls** could be substantial. An obvious optimization was to allow the user process to issue many operations with a single **ioctl** call.

To make this easier, we have developed a set of macros which allow a user process to build a small script containing **alloc, copy, send, free, wait_rx** and **rx** operations, and pass it via **ioctl()** to the Jetstream driver where it is executed operation by operation. Results of individual operations are placed within the script and the modified script is copied out as the **ioctl()** terminates. Currently, there is an arbitrary limit of 64 operations per script.

We have measured the performance improvement that is obtained when operations are collected into a script. Table 2 shows the throughput obtained by a simple byte-stream protocol when it uses one operation per call and when it uses scripts. It is clear that the use of scripts yields about a 30% performance advantage over one operation per call. Note, a typical script contains between 20 and 40 operations.

Message size (bytes)	Throughput (10^6 bit/s)		
	Script	1 op per call	Improvement (%)
4096	195.0	146.6	33
8192	195.1	148.3	31
16384	194.5	148.3	31
32768	182.1	132.6	27

TABLE 2. The impact of system calls

We do not expect application writers to use scripts directly; instead we believe scripts will be constructed by special libraries which export simpler interfaces. We have developed two experimental libraries in order to evaluate the performance that user processes might expect to achieve.

The first of these is for applications requiring datagram or RPC services. It allows datagrams up to 64 Kbyte to be sent and received. The library provides a **send** function, a **recv** functions and a **rpc** function which performs both a send and a recv. We have measured the performance that user-space applications obtain with remote procedure call operations based on our library as well as the standard UDP services. Table 3 shows the time for each transaction as we vary the amount of data that is sent and received at each call. Our library interface is around 30% more efficient than UDP.

The second experimental library is for applications requiring reliable byte-stream services. It provides connection setup functions as well as the normal send and recv functions. It is similar to TCP insofar as it uses cumulative acknowledgements and a sliding-window flow-control scheme and exhibits comparable behaviour in a LAN environment. In section 5 we compare the performance of our user-space library with that obtained using the kernel TCP. The measured results show that our library obtains performance levels very close to those of the single-copy version of TCP and substantially better then the two-copy TCP. We note that our stream library was coded and debugged in about four weeks and is still being tuned, so better results may be forthcoming.

RPC Data size Send, Recv (bytes)	Round Trip Time (µs)	
	Library	UDP
16,16	322	471
32,32	324	476
64,64	328	496
128,128	333	495
256,256	346	592
512,512	384	663
1024,1024	449	742
2048,2048	589	924

TABLE 3. RPC Performance

User-space protocol implementations have a number of well-known advantages [19]. For example, they ease the prototyping and debugging of new protocols, and allow application specific knowledge to be exploited to allow performance improvements to be made. We believe our results show that with the right low-level interfaces, user-space protocol implementations can also achieve the sort of performance normally associated with kernel-based protocols.

4.4 Implications for User-Space Protocol Developers

It is clear from the results in Table 2 that operations on user pools need to be batched to achieve the highest performance. In particular, libraries that implement byte-stream protocols need to batch operations together in an efficient manner.

Fortunately, byte-stream protocols by their very nature lend themselves to efficient pipelined operation. For example, consider implementing TCP using the pool operations described above. If we view the TCP sequence space from the send side, a natural pipeline is obvious: at the front of the sequence space new blocks are being **alloc'd**; further back in sequence space allocated blocks have data **copy'd** into them and blocks that are full (with checksums available from the **copy** operation) are **sent**. At the trailing edge of sequence space blocks are **free'd** when acknowledgements are received. Using the Jetstream primitives, all of these operations, including TCP ack reception, may be accomplished in a single batch.

Whilst we have not yet implemented a user-space TCP on top of Jetstream, we are convinced that a TCP implementa-

tion that batches the primitive operations in this way would perform at least as well as a kernel implementation of the protocol.

4.5 General Observations

We observe that the scheme we have described for the Jetstream adapter need not be limited to network device drivers. It might prove beneficial for other device drivers to make their primitive operations visible to user processes. For instance, consider a disk driver that allowed user code to **open** files and directly **read** or **write** blocks of the file. If disk device scripts and network device scripts could in some way be amalgamated, it would be possible for a user program to form a script to copy files straight onto the network, or from the network straight to disk. The key point to note in these scenarios is that the user process is in complete control of a data stream, without the data ever needing to cross the kernel-user space boundary.

5 Performance

In this section we first show how throughput is affected by the maximum Jetstream frame size and the Afterburner buffer size. We then present application-to-application throughput measured over Jetstream using TCP (single- and double-copy) and our own reliable byte-stream protocol.

The measurements reported here were collected from two HP 9000/735 workstations running HP-UX 9.01 and using the **netperf** [20] utility. TCP window scaling [21] is used with socket buffers of 245760 bytes. Both workstations are connected to the site Ethernet and have the usual background processes. Note that the sink **netperf** process receives the data but does not examine it, i.e. the data is copied to main memory but not brought into cache.

5.1 The effect of the maximum frame size

Table 4 shows the measured throughput in Mbit/s as we vary both the maximum frame size and the size of the Afterburner buffers. The frame sizes are chosen to be multiples of 4096 bytes plus space for various headers.

These results show that increasing the frame size yields diminishing returns. This is what we expect: the per-frame overhead is already quite small when we use a 32 Kbyte frame, so doubling the frame size does not show much improvement. It is quite reassuring that we do not need to have very large frames in order to achieve good performance; it would appear that even 16 Kbyte frames provide adequate results. Similarly, increasing the Afterburner block size does not yield large improvements and even 8 Kbyte blocks yield throughputs in excess of 200 Mbit/s.

Frame size (bytes)	Throughput (10^6 bit/s)		
	4K block	8K block	16K block
8256	150	-	-
16448	173	184	-
32832	187	200	-
49216	192	206	-
65344	-	210	217

TABLE 4. Measured TCP performance

It is clear that a single workstation cannot use the full network bandwidth: the current bottleneck is the rate at which the processor can move data across the graphics bus. Although a single pair of workstations achieve 200 Mbit/s, we hope to obtain results for three simultaneously active workstations in the near future to confirm that the aggregate network performance can be much higher.

5.2 Throughput of byte-stream protocols

The next set of results show how important it is to reduce data copies to a minimum and they also show that protocol implementations do not have to be kernel-based in order to achieve high throughput.

Figure 3 shows application throughput for three configurations. In all three we have measured the throughput as the application message size is increased. The maximum frame size was set to 61504 bytes, the Afterburner block size was 4096 bytes and 245760 byte socket buffers (or equivalent) were allocated.

The first and second configurations use the **netperf** application with the kernel TCP. One configuration uses the single-copy TCP and the other uses the two-copy implementation. It is very clear that reducing the number of copies from two to one has a dramatic effect on the throughput. Another result from this graph is that the application does not need to use large buffers in order to achieve good performance - the knee in the curve occurs with message sizes of only 2 Kbyte.

The third set of results show the performance of a **netperf**-like application that uses our reliable byte-stream protocol. The objective was to show that if the network driver provides the right interface then protocols can be developed and run in user space and still obtain high performance. These results confirm this as we see that the user-space protocol can achieve throughput that is comparable with the kernel single-copy TCP.

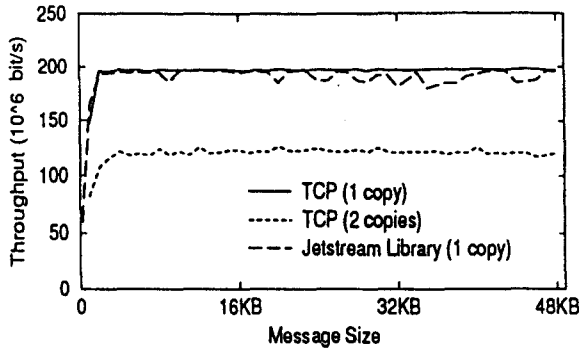

FIGURE 3. Performance of Reliable byte-stream protocols

6 Conclusions

High speed networking faces two important issues. The first is to find cost-effective Gbit/s technologies that can support both new delay-sensitive applications together with existing ones. The second issue is that the driver-level interface between the network and the application must become much more flexible to support these applications; in particular, it must provide applications with greater control over how incoming data is processed.

We have addressed the first of these issues with the development of an experimental LAN, called Jetstream, which provides Gbit/s link rates as well as the ability to provide guaranteed bandwidth and tightly controlled access delays. With Jetstream we have eschewed the move towards switch-based networks because we have focused on providing a very low-cost system. Further, the use of variable size frames means that we avoid the costs of segmentation and reassembly, while the high link rate can assure low access delay. Jetstream frames use the same format as ATM cells and this provides two important benefits; the first is that interworking between Jetstream and ATM networks becomes very simple; the second benefit is that the channel identifier provides fine grain control over application data.

The presence of the channel identifier in every frame means that the Jetstream driver can immediately associate an incoming frame with its consumer. We have exploited this with the development of a driver that enables applications to control how their data should be managed without the need to first move the data into the application's address space. Consequently, applications can elect to read just a part of a frame and then instruct the driver to move the remainder directly to its destination.

We have built a small number of prototype Jetstream interfaces, and these have been used in conjunction with our Afterburner card and our single-copy version of TCP to provide application throughput in excess of 200 Mbit/s. A Jetstream driver has been developed which uses the Afterburner buffer memory to associate 'pools' of buffers with application channels. Applications control how their buffer pools are managed by the use of simple operations which are combined to form control scripts. We have implemented a reliable byte-stream protocol using such scripts to show that protocols can be developed and run in user-space and yet still achieve levels of performance comparable with kernel-based implementations.

We conclude that low-cost, high-performance Gbit/s networking can be practical. Such a network can provide the services and flexibility that will be needed to support not only the emerging delay-sensitive multimedia applications, but also will enable current applications to achieve much better performance than they do today.

Acknowledgements

We would like to take this opportunity to thank Steve Pink for encouraging us to write this paper.

References

[1] G. Watson, D. Banks, C. Calamvokis, C. Dalton, A. Edwards and J. Lumley, 'AAL5 at a Gigabit for a Kilobuck,' to appear in *Journal of High Speed Networks*.

[2] 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.

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

[4] K.C. Sevcik and M.J. Johnson, 'Cycle-Time Properties of the FDDI Token Ring Protocol,' *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, pp. 376-385, March 1987.

[5] CCITT, Blue Book, Vol. 3, F3.1, Recommendation G.114, 'Mean one-way propagation time'.

[6] R.C. Walker, T. Hornak, C.-S. Yen, J. Doernberg and K.H. Springer, 'A 1.5 Gbit/s Link Interface Chipset for Computer Data Transmission,' *IEEE J. Select. Areas in Comms*, Vol. 9, No. 5, pp. June 1991.

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

[8] I. Cidon, I. Gopal, P.M. Gopal, J. Janniello and M. Kaplan, 'The plaNET/ORBIT High-Speed Network,' IBM Research Report 92A005472, August 1992.

[9] P.A. Steenkiste, B.D. Zill, H.T. Kung, S.J. Schlick, J. Hughes, R. Kowalski and J. Mullaney, 'A Host Interface Architecture for High-Speed Networks,' In *Proceedings of 4th IFIP Conference on High Performance Networking*, A. Danthine and O. Spaniol (Eds.), pp. A3-1 - A3-16, December 1992.

[10] B.S. Davie, 'A Host-Network Interface Architecture for ATM,' in *Proceedings SIGCOMM 1991*, Zurich, Switzerland, pp. 307-315, September, 1991.

[11] Van Jacobson, 'Efficient Protocol Implementation,' *ACM SIGCOMM '90 Tutorial*, September 1990.

[12] D.M. Banks and M.J. Prudence, 'A High Performance Network Architecture for a PA-RISC Workstation,' *IEEE J. Select Areas in Comms.*, Vol. 11, No. 2, February 1993.

[13] D.D. Clark, Van Jacobson, J. Romkey and H. Salwen, 'An Analysis of TCP Processing Overhead,' *IEEE Commun. Mag.*, pp. 23-29, June 1989.

[14] C. Partridge, 'Gigabit Networking,' Chapter 9, Addison-Wesley Publ., 1993.

[15] Jeffrey C. Mogul, Richard F. Rashid and Michael 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.

[16] O. Hagsand and S. Pink, 'ATM as a link in an ST-2 Internet,' in *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 189-198, Lancaster, November 1993.

[17] C.B.S. Traw and J.M. Smith, 'Hardware/Software Organization of a High-Performance ATM Host Interface,' *IEEE J. Select Areas in Comms.*, Vol. 11, No. 2, Feb. 1993.

[18] P. Druschel, L.L. Peterson, B.S. Davie, 'Experiences with a High-Speed Network Adaptor: A Software Perspective,' in *Proceedings SIGCOMM 1994*, London, September, 1994.

[19] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy and Edward D. Lazowska, 'Implementing Network Protocols at User Level,' in *Proceedings SIGCOMM 1993*, San Francisco, pp. 64-73, September, 1993.

[20] 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 ftp.csc.liv.ac.uk in /hpux8/Networking

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