Engineering a User-Level TCP for the CLAN Network

Position paper

Kieran Mansley
Laboratory for Communication Engineering
University of Cambridge
Cambridge, England
kjm25@cam.ac.uk

ABSTRACT

As networks and I/O systems converge and the bandwidth of networks increases, conventional approaches to networking are struggling to deliver the performance and flexibility required.

CLAN (Collapsed LAN) is a high performance user-level network targeted at the server room. It supports RDMA and programmed I/O (PIO). We have implemented a set of IP based protocols at user level, and shown how true zero copy transmission (without modifying the sockets API) and reception can be achieved.

In this paper we discuss the problems associated with placing protocol stacks at user level and the architectural decisions required to obtain high performance. We also introduce our work using the network gateway which connects CLAN to the Internet to assist a server cluster in protocol processing.

1. INTRODUCTION

The line speed of local area networks has increased by orders of magnitude in recent years. As it reaches a gigabit per second, the network itself is often no longer the bottleneck in transferring data from one host to another. Instead, the overhead of moving the data between the application and the network [25] and performing protocol processing [23] has become critical.

The overhead of traditional networking is due to a number of factors [11] including copying data between buffers, demultiplexing, interrupts, system calls, and inefficient protocols. These use up CPU cycles that could be doing useful work for applications.

Networks are starting to be used in a number of unconventional ways, and the roles of networks and storage are converging. For example, iSCSI [22] is an emerging standard aimed at Storage Area Networks. It allows SCSI commands to be issued over TCP/IP to remote devices. This presents problems for the conventional structure of operating systems. Each request for data by the application must go through two stacks (filesystem and network) in the kernel and there is a dependence between them. This leads to a more complex implementation and reduced performance. To perform an iSCSI operation requires many more CPU operations than an equivalent SCSI one. To address this problem there have been attempts to perform TCP/IP on a processor on the Network Interface Card (NIC), and have it present a SCSI interface to the operating system. This dramatically increases the cost of the network hardware. McAuley and Neugebauer [24] suggest using virtual machines as an alternative to increasing the number of processors.

The recent IETF draft proposals for Remote Direct Data Placement (RDDP) and Remote Direct Memory Access (RDMA) [5] describe another unconventional way of using networks. This style of transfer is becoming increasingly important in high performance networking, and has again motivated the move toward placing processors on NICs. Co-processors (while attractive for specialised operations such as graphics) are being increasingly deployed in I/O systems. Although this removes load from the main CPU it may not be beneficial in the long term as co-processors continue to lag behind the speed of CPUs. Co-processors also add latency to the data path in the NIC.

User-level networking has the potential to address many of these problems. In this style of networking, the application is able to communicate directly with the NIC, bypassing the operating system for the majority of operations. Similarly the NIC is able to deliver received data directly into the application's buffers. We have developed a user-level network which provides an API with similar semantics to the above IETF draft standards. We have also used this network to implement a suite of protocols at user level.

The IP suite of protocols (including TCP, UDP, ICMP) are used for many Internet communications, and it is important that they perform well. Despite a significant amount of research into efficient implementations of user-level protocol stacks [21, 31] there are still many areas that have not been fully resolved.

In this paper we describe how we have addressed the issue of providing efficient protocol implementations for use in innovative networks. In particular we have created a high performance suite of IP based protocols for the CLAN network. Our work is focused on TCP/IP in a server cluster network and bridging this to the Internet. Section 2 gives an overview of the hardware and key software abstractions that the CLAN network provides. In Section 3 we describe the architecture and structure of our user level protocol stack. Section 4 describes our approach to user level reception, while Section 5 deals with user level transmission and introduces how we have developed a system to use the gateway to assist the cluster nodes in protocol processing. Finally we out-line how we plan to measure the performance of this setup and highlight future work.

2. THE CLAN NETWORK

CLAN is a low latency, high performance, user-level network. It has a raw bandwidth of 1Gbps. Its primary targets are the server room and cluster computing.

In the rest of this section we provide a brief introduction to CLAN. In previous work Riddoch et al have published detailed descriptions of the hardware and software support, as well as Tripwire [30] (the synchronisation mechanism), and its use to support a variety of protocols and applications [28] including VI [29].

2.1 Low Level Data Transfer

At the lowest level CLAN is a Distributed Shared Memory (DSM) interface. Regions of memory can be mapped from one host across the network to another allowing very low latency transfers using standard processor write instructions. In this way it is similar to the SHRIMP [6] network which uses reflective memory. In addition to Programmed IO (PIO) the CLAN NIC also provides a DMA engine to allow longer transfers to be offloaded from the CPU.

The format of data packets on the wire is similar to write bursts on a memory bus. The packet consists of a start address in memory, followed by the data to be written to that address. There is no length field. This is an interesting property, particular when it comes to switching; packets can be arbitrarily split or merged part way through transmission (as you do not need to alter the header). The switch is therefore able to prevent long packets hogging contended ports. Small packets can be combined in the network to reduce overheads (this is particularly likely to happen when congestion occurs, so increasing efficiency and relieving the congestion). It also enables NICs to start transmitting as soon as data is available, without waiting for an entire packet.

The prototype network does not provide any security against malicious code writing to apertures that belong to other endpoints. This would be essential in a commercial implementation, and a solution similar to that developed for Hamlyn [37] could be used.

The API for CLAN takes a different approach to other user-level networks [7, 36, 32]. It presents a single, low-level network interface that supports communication with low overhead and latency, high bandwidth, and efficient and flexible synchronisation. More complex interfaces can be built on top of this without considerable additional overhead. It is implemented using simple hardware without on board processors.

Although at the lowest level CLAN is a DSM network, it is not intended that the normal DSM style of communication is used by applications. Instead, the DSM support is used as the base for building higher level communication abstractions.

2.2 Distributed Message Queue

One of these abstractions is the Distributed Message Queue (DMQ) as shown in Figure 1. It is essentially a flow controlled messaging abstraction, and is described here in its simplest form.

A DMQ is similar to a circular message queue with two pointers, one to indicate the current read position (`read_i`), the other to indicate the current write position (`write_i`). Both the sender and receiver keep a "lazy" copy (in the shared address space) of the pointer they are *not* responsible for. The buffer for the circular queue physically resides in the memory of the receiver and the sender has a mapping of it in its own address space. By writing packets to these mappings and updating the queue pointers the two nodes can communicate.

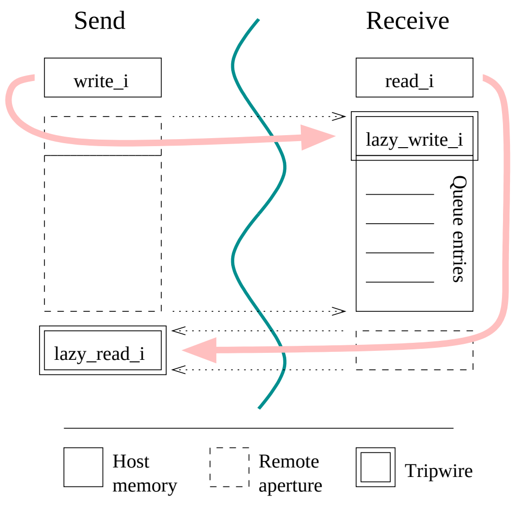


Figure 1: A Distributed Message Queue

To perform transfers in this way requires only a few processor write instructions. As a result it represents very low overhead. The amount of physical memory required for the buffer is also small (around 10KB for full Gbps throughput) due to the low latency.

Synchronisation is performed using Tripwires [30] which provide a low-overhead mechanism for notifying the application of changes to the queue.

This user-level API can also be applied to other server room interconnects. In particular we are working with a Gigabit Ethernet based system called EtherFabric from Level 5 Networks Ltd. [1] This new hardware should allow easier implementation and has the potential for further interesting experiments with the technology described in the rest of this paper.

2.3 CLAN Hardware

We have a prototype hardware implementation consisting of a number of Network Interface Cards (NICs), two 5-port switches, and a bridge (between CLAN and Gigabit Ethernet) in development. The current hardware has some weaknesses (for example the DMA engine only allows a single request at once, and generates an interrupt after each transfer) but represents a viable platform for research into software support for user-level networking. These weaknesses will hopefully be solved by using the new hardware available from Level 5 Networks.

While the hardware used is proprietary, it is all fabricated using cheap off the shelf components, and as a result would compare favourably to existing Gigabit Ethernet NICs in terms of cost when produced in volume.

The bridge was still under development when AT&T Laboratories Cambridge Ltd closed in April 2002. As a result, it is currently unfinished. To allow bridging experiments to continue we are using an Intel STL2 dual processor server PC equipped with CLAN and Gigabit Ethernet NICs to perform this role. A new version of the NICs designed to run at 3Gbps was also in the pipeline when the laboratory closed.

All the hardware used by CLAN is simple and lightweight compared to other similarly performing networks. This results in a more scalable implementation. Because there are few on board resources used by each endpoint (Tripwires being the only one) and no on board processor, the hardware itself does not impose as many limits on the number of concurrent connections as other technologies. Co-processors on NICs results in a more complex data path, and as network speeds are currently increasing by orders of magnitude every few years (outstripping the increase in speed of specialized processors) this is likely to become more critical.

3. STACK ARCHITECTURE

Traditional kernel protocol stacks are executed in a different context to the application they are serving. The large overhead associated with context switching is one of the primary factors that motivated the move to user-level networking. However, initial attempts at developing user-level network stacks have used a similar architecture to their kernel ancestors [8]. To ease implementation (many user-level stacks are direct ports of kernel stacks [26]) the protocol processing generally occurs in a separate thread to the application. This has a number of disadvantages. Firstly, although you have exchanged context switches for thread switches these are both considerably more expensive operations than a function call. Secondly, protocol processing is done at some undetermined time after an application issues a request to send or receive data (at the mercy of the scheduler), and this can lead to artificially increased latency. TCP's window size is sensitive to latency, so by acknowledging in a timely manner you will increase the window, and increase the throughput.

Although separate threads for different tasks make dealing with multiple connections, timers, etc, considerably easier, it was decided for the CLAN user-level TCP stack to attempt to do the majority of protocol processing in the same thread as the application.[1]

The CLAN TCP/IP suite is based on lwIP [14, 15], a lightweight implementation of IP, TCP and UDP. It is designed for low-memory systems, such as embedded processors. We have heavily modified it to support high performance rather than its design goal of low memory usage. In particular the threading model has been changed. lwIP was chosen for its clean and simple code base which easily adapted to our needs. This has proved considerably easier than taking a higher performance stack (such as the Linux kernel stack) and attempting to re-architect it at user level.

lwIP has a linear model for its threads as shown in Figure 2. There is a thread for the application and sockets interface, a thread for the TCP/IP stack, and a thread for the network interface. Data must pass through each of these threads when either sent or received. It can function in an operating system without thread support, but in this case it cannot use the sockets API and it still requires some external path of execution to call its timer and incoming packet functions at the appropriate times. Because all TCP/IP processing is done by one thread, all accesses to the TCP/IP stack are serialised through the use of message queues, which require semaphores to provide coherency.

For the CLAN TCP/IP suite this has been adapted so that all protocol processing and network card access is performed in the

[1]This approach is becoming popular in other areas of computing where high performance is required. For example omniORB [33] avoids thread switches on the call path by performing all work in the calling thread.

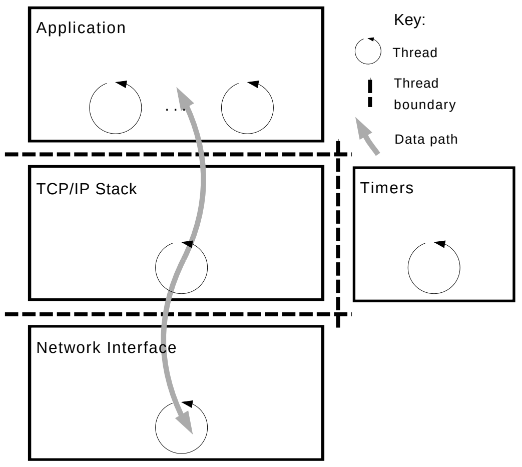


Figure 2: lwIP TCP/IP Architecture

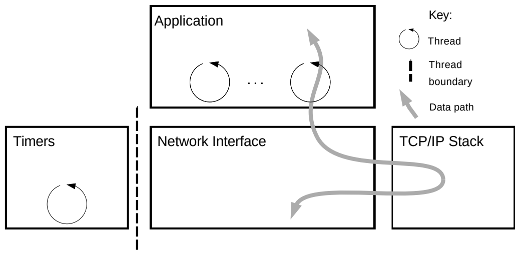


Figure 3: CLAN TCP/IP Architecture

same thread as the application (with the exception of timers, which have a separate thread; we are currently investigating how TCP timers can be more efficiently implemented). As a result, the data path has no thread switches. As each application thread can access the TCP/IP stack directly without having to go through a semaphore controlled message queue there are fewer locking overheads (although some effort had to be expended to ensure the TCP/IP code was thread safe). This has lead to an architecture as illustrated in Figure 3 where rather than the components being arranged in a linear fashion they are arranged with the CLAN network code acting as a hub. The TCP/IP stack, instead of being the means by which the application accesses the network (via the sockets API) is now a tool for the network interface to use, and the application accesses the CLAN network code directly (but still via the sockets API).

To support this change in the way protocol processing activity is driven does not require any modifications to the application, other than to link against a different shared library. (Some modifications are required however to achieve the separate issue of zero copy reception of data as described in Section 4.2). A common criticism of other user level network libraries is that applications cannot use `select()` with a combination of the user-level socket file descriptors and traditional OS file descriptors. In our case this is possible due to the way the asynchronous event queues that select responds to are implemented (see [30, Section 3.5] for further details), and is invisible to the application.

A good example of the implications of the change in threading to the stack is the implementation of blocking reads and writes. In normal circumstances these would block at the thread interface between the application/sockets API and the TCP/IP stack. For example, writes would block waiting for space in the TCP send queue. By removing this thread boundary we are no longer able to block in this way. Instead, because the stack is executing in the same thread as the application, we use the processor time that would otherwise have been released (due to the application blocking) to perform the protocol processing. This can contribute to reduced latency as protocol processing occurs as soon as something is queued for writing rather than when the TCP/IP stack thread is next run. For receives, protocol processing is done lazily (i.e. when the application asks for it). This should result in improved cache performance as the data is touched by the stack just before the application makes use of it. The advantages of this technique have been demonstrated by Druschel & Banga [13].

Implementing this change in architecture has been interesting. Often changing assumptions about the way things are organised is a good way to expose the limitations and fragility of code. However, the stack we have chosen has coped very well with this ordeal. The most interesting problems encountered have been:

Connections. Having a connection oriented network (CLAN) beneath a connectionless protocol (IP) brings us advantages in demultiplexing, but in turn presents its own problems. For incoming packets we have more knowledge than is usual (because we know which connection the packet arrived on) and we need some way to propagate this knowledge into the protocol stack so that it is not deduced again in the normal way. Similarly, for outgoing packets the application layer knows which socket a write has occurred on, and propagating this knowledge to the network layer is helpful. We have provided hooks into the data structures that track each packet to allow this information to be passed. To complicate matters there are also numerous special cases (e.g. a reset sent by the TCP stack) for which there is no mapping to a socket.

Understanding all of the interfaces involved. The protocols are generally well documented, but the sockets interface has evolved over many years. Its documentation (understandably) focuses on how to use it in the simple case, rather than how to understand all the different ways it can be used in and the significance of the details.[2]

Optimising the common case. Making the common case (data reception and transmission) fast is good, but it can result in increased complexity for less common operations. To allow us to judge the overall benefit of a change we have developed a profiling system to graphically compare the time taken to do a particular operation in two (or more) different implementations.

API to the network interface. In traditional architectures the API between IP and the network interface is simple (in essence, one function to call to transmit data, and another to call when data has been received). We have changed the architecture to make the network interface code a hub for all communication with the application. As a result the code must now provide a much richer interface and make this accessible to both the application (through sockets) and the protocol stack.

[2]Just like footnotes, "The devil is in the detail".

4. USER-LEVEL DELIVERY

One of the most important differences between traditional networking and user-level networking is the way incoming data are delivered to the application.

In the kernel-based architecture incoming packets are delivered to a pool of packet buffers, which are then examined by the kernel to determine the application for which they are destined, and queued waiting for the application to perform a read. The data must be copied from the kernel packet buffers to the application memory space.

User level networks have taken a variety of approaches to delivery. The most difficult part is the one that was performed by the kernel - that of demultiplexing the incoming packets; i.e. determining which application it should be delivered to. Some user-level networks have left this functionality in the kernel [35, 16]. Some have even chosen to leave IP in the kernel [8], but in doing either of these they take a large performance hit compared to a pure user-level network. Others have chosen to implement this (and possibly other) functionality in the NIC itself [26, 7], but this requires more complex (and therefore expensive) hardware. Also, as the NIC must store state for each connection, the available hardware resources place a limit on the number of concurrent connections that can be supported.

In our implementation we were keen to avoid both of these pitfalls, and this is simplified by the physical network. A transfer within a CLAN network is analogous to a write burst on a memory bus, or an RDMA Write. Each write consists of a start memory address where the first word should be written, and is followed by the data. This means that the network is send-directed, whereas the majority of others are receive-directed; i.e. it is the sender that determines the final location (in the receiver's memory) of the data, not the receiver. This makes the receiver's role in the demultiplex much simpler. However, in order to ensure the data ends up in the correct place the receiver must inform the sender where the data should go in advance. (This is performed as part of connection setup). This style of transfer has recently been proposed as a draft standard for Remote Direct Data Placement (RDDP) and Remote Direct Memory Access (RDMA) by the IETF.

4.1 Implementation

The model used to transport IP over CLAN is built around a structure similar to the Distributed Message Queue discussed in section 2.2. A circular queue is shared across the network, with the remote host writing data, and the local host reading data. There is one DMQ (or more) per socket. For TCP/IP there are essentially two operations that need to be performed on each packet.

- Firstly, it must be processed by the relevant protocol stacks.
- Secondly, if it contains valid data, it must be passed to the application.

As a result the queue in this case requires three pointers, rather than normal two (read and write). The write pointer is the same as before, but we now subdivide "read" into a protocol pointer and a delivery pointer, which keep track of the respective tasks' progress.

In this way, we are able to perform both delivery and protocol processing directly on the data, in place, without copying it. It also separates the act of protocol processing from the act of delivery.

We also separate the headers from the payloads and transfer them into two separate queues. In this way we prevent the payload queue becoming fragmented as headers are processed and discarded in advance of the application reading the payloads. It also allows for some optimisations on the headers queue as the entries are all fixed size and small enough to be efficiently transferred using PIO instead of DMA. Synchronisation is performed on the headers queue.

Should an out of order segment be received it is copied out of the DMQ to prevent it causing delays for in order packets. This copy prevents additional complexity on the fast path for received data. It is subsequently processed in the normal way.

As the application is able to access the queue memory directly, we need some mechanism to disguise the circular structure of the queue. To do this, we use a Virtual Ring Buffer [19], where the physical pages that make up the queue are mapped into two adjacent pages in virtual memory. The application (or TCP/IP stack) is then able to read the entire contents of the queue starting from any position within it. Therefore, packets which wrap round from the end to the beginning do not require special treatment, and no alterations to the application or stack are necessary.

4.2 Extensions to preserve data

A copy is also necessary unless the sockets API is modified to return a buffer with the received data in, instead of taking a buffer from the application and filling that with the data. In this case, should a server application wish to preserve the received data it will still need to copy it out of the queue to make space for more incoming packets. There are two approaches that can be taken to freeing space in the queue. The less robust method is to assume that when the application next performs a read operation on that socket, it has finished with the data from the previous read, and we can now release that space in the DMQ. Alternatively if this assumption is not valid the sockets API also requires modification to add a call for the application to signal that it is now safe to reuse that space. However, for many server applications this assumption is valid as the stream of incoming data is only processed once and then immediately discarded or written to a more permanent medium.

If the application knows it will need to preserve the data in memory, and it does not wish to take the performance hit of copying the data, it can publish an area of memory that it would like the payloads to be delivered to. Then, instead of the payloads going into a circular queue they can be delivered directly to the application's chosen location. The headers are removed and processed in a circular queue as before.

Although more efficient for some servers, this does require some changes to the application and modifications to the sockets interface in order to publish the buffers in advance. Similar extensions were made to the API for WinSock2.

5. ZERO COPY TRANSMISSION

So far we have only examined the changes we have made to support zero copy *reception* of data. Zero copy transmission has a different set of associated problems. Existing attempts at zero copy TCP [10, 9] have either modified the sockets API, or used copy-on-write page flipping.

The sockets interface was not designed to perform zero copy operations. When `send()` is called it returns once the data has been placed on TCP's send queue. There is still a considerable amount of protocol processing to be done before the packet is transmitted, and it could be some time before it is safe to discard the data. As a result the stack is required to copy the data to prevent it being overwritten by the application. Even if this were not the case and the buffer could be guaranteed until the data are written, many high performance network cards use DMA to transfer the data to the network and the DMA request may not happen immediately. The data must be preserved until the DMA has completed. Even if you achieve this without copying, you still need to preserve the data until a TCP acknowledgement is received in case you are required to retransmit. This will take at least one round trip time (RTT) and when communicating over the Internet RTTs can be large.

In the rest of this section we discuss a number of ways to address this issue.

5.1 Update the Sockets API

Firstly, because the problem stems from a weakness in the sockets interface the obvious choice would be to remedy this by changing the interface. It needs some way to notify the application when it is safe to re-use the buffer. Many zero-copy APIs have been developed [31] to do just this, including the IEEE standard for *Asynchronous I/O* [20]. Many would argue in favour of this option; we should not be expending effort to provide workarounds for outdated APIs. However, while that might be an ideal, in reality, if there is a solution that (all else being equal) can fix the problem without changing the API, it will be more popular. The resistance to changing APIs should not be underestimated.

5.2 Alterations to the transport protocols

We could also argue that retransmits by servers are, in many cases, unnecessary. Were this burden removed from the server the need to wait for an ACK would go away. Where the data being served are static, the client could simply re-request any erroneous or missing data, especially if Application Level Framing [12] is used. Where the content is dynamic the client would have to ensure that it received a coherent set. There may be cases where this is not feasible and so, in those at least, retransmits are desirable.

This could be realised by using UDP as the transport protocol in place of TCP and many modern servers and clients (for example RTP or NFS) effectively do just that.

5.3 Blocking semantics

The application will (or at least can) re-use the buffer as soon as the `send()` returns. Therefore, if we do not want the buffer to be re-used we do not return from the `send()` call. Instead of blocking until the data have been queued on the TCP send queue, we could block until the data have been sent, or the DMA operation has completed, or even until the TCP acknowledgement has been received. However, blocking until a TCP acknowledgement is received for each `send()` in the worst case reduces TCP from a streaming protocol to a ping-pong protocol. Only one `send()` per socket can be in-flight at any one time.[3] Because of the single threaded nature of our TCP stack it is easy to experiment with this change in blocking behaviour. It was observed that moving from a system where it blocks until the DMA has completed to a system where it blocks until an acknowledgement is received results in at least half the throughput.[4] It is clear this is not suitable as a

[3]This is not the same as a single *packet* being in-flight at once, as applications often pass large blocks of data in a single write operation.

[4]The exact degradation will be strongly related to the ratio of

solution, but blocking the `send()` until the DMA has completed may be justifiable. This would solve the problem for transport protocols such as UDP where retransmits are not supported.

If we do this, we still need some way to guarantee access to the data for protocols which do perform retransmits. We have started to look at ways that the data could be copied with low overhead. These are all based around performing the copy at the same time as writing the data to the network, and so absorbing some of the overhead of the copy into the network write. (They are essentially very similar operations, especially in a distributed shared memory based network such as CLAN).

5.4 Memory Multicast

Essentially what we are trying to perform is “memory multicast”. We would like to write data to the network’s memory mapping and to a physical memory mapping in the same operation. Without support for this from the hardware it is difficult to implement directly, but we have considered a number of ways in which existing hardware may be used to achieve this effect.

One of these is based around abusing the Intel architecture when PIO is used[5]. The Intel Pentium Pro (and subsequent) cores feature a RISC like architecture. x86 instructions are executed by breaking them into smaller, simpler operations called micro-ops. These micro-ops are then placed into a pool and can be scheduled out of order to achieve more optimal use of the CPU’s resources. There are five execution units to carry out these micro-ops, and so they can be executed in parallel. As a result, if x86 instructions to output the data from memory to the network, and x86 instructions to copy the data from memory location to another are interleaved, there is a reasonable chance that these will be optimised to occur in parallel. However, both writes will need to traverse the front-side bus, and this could limit the performance. Also, as the micro-ops have no visibility outside the CPU the programmer has no control over the order of their execution, and in any case it relies on this particular implementation of the x86 architecture. As a result, we have chosen not to attempt to implement this suggestion, but it should be seen as an illustration of how, when performing one write, doing another at the same time may be less than twice as expensive.

5.5 Write-back from NIC

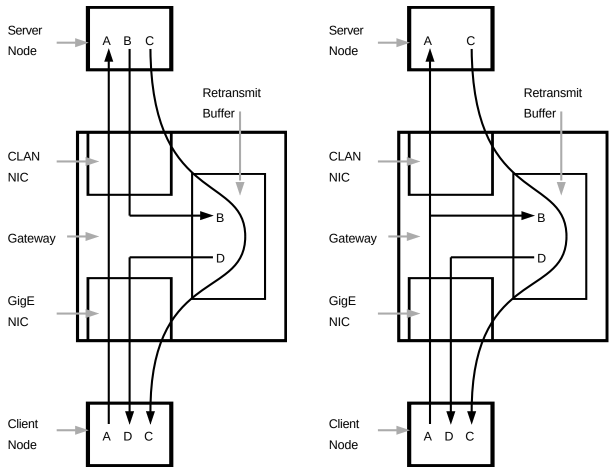
The NIC could write outgoing packets back to memory as well as to the network. While this would not be limited by the resources available on the NIC, it would clearly double the PCI and memory bus usage. It would also be hard to schedule it so that you had nice bursts in both directions. In particular when using PIO to write data to the network, the outgoing writes would take priority over the incoming ones, and the NIC would soon run out of FIFO space. As a result, this is not an acceptable solution.

5.6 NIC Assisted Retransmission

A more feasible solution would be to add an expansion card to accompany the NIC. This would be relatively simple, consisting of an FPGA, some memory, and a PCI bridge. This card would snoop the PCI bus, watching for writes to the network. It would store the data associated with such writes in its RAM. (While not used for retransmission, a similar approach [34] has been suggested for related problems.) Should a retransmit be necessary the stack could then request the expansion card DMA the data to the NIC, and in so doing perform the retransmit. This achieves preservation of the data with no additional CPU load and no additional bus bandwidth consumed, but it does have a number of disadvantages. Firstly, there is the expense of the extra hardware required. This could be reduced by integrating it into the NIC but it would still dramatically increase the hardware costs and limit scalability, especially for a simple card like the CLAN NIC.

5.7 Copy On Write

A different approach would be to only copy the data if necessary, and avoid it in all other cases. We only need to copy the data if the application tries to re-use one of its recently used buffers. (Many servers will be dealing with static content, and so may never do this.) It would be straightforward to implement a copy-on-write system where when a buffer was passed to the stack that page was marked as “in-use” until the corresponding TCP acknowledgement is received. If the application tries to write to that page in the meantime the page would be copied. Clearly this requires manipulation of the page table entries by the protocol stack. This would be easy if the stack were in the kernel (and in fact this has been used by kernel stacks [17]), but from user-level we would either have to make a system call (which would negate much of the benefit of having the stack at user-level) or we would have to perform user-level memory management and page table manipulation, which in turn would need selective TLB flushing. It would also only give benefit to those applications who normally would not require the buffers to be copied. While many may fall into this category, there are some notable exceptions (particularly benchmarks) which would not. In these cases you would be worse off than if you had just copied the data to start with because of the additional overhead in page table manipulations.



(a) Retransmit requested by server

(b) Retransmit requested by gateway

A is the incoming data path, *B* is the retransmit request path, *C* is the outgoing data path, and *D* is the retransmitted packet path.

Figure 4: Gateway Assisted Retransmission.

bandwidth to latency of the network.

[5]Rather unusually we are using PIO to write data to the network for small transfers as this gives much higher performance than DMA

5.8 Gateway Assisted Retransmission

The underlying CLAN network is more reliable than current technologies such as Gigabit Ethernet. This is facilitated by the property that you cannot write data to a host unless the host has space to receive it. i.e. Packets are never dropped by the receiver. Many networks do not have this property. The new hardware from Level 5 Networks will also have error detection support built in. This suggests a solution that does not consume any additional server resources. To connect the CLAN network to the wider world we require a bridge. Part of our research has been investigating how the bridge can be used as a protocol processing gateway to help with TCP processing.

Rather than require the server node to preserve the packets it has transmitted until they are acknowledged, the gateway could do this more efficiently. The gateway must access all data flowing out of the network, and so can buffer it in memory at little extra cost to itself. As a result, a copy of the data is preserved by the cluster without any extra CPU, bus, NIC resources, or network bandwidth being used on the server node.

There is the additional cost involved at the gateway, but this is less critical than the cost of the NICs (as there are far fewer gateways than NICs). Also, by having the retransmit memory shared by all the nodes in the cluster we can make more efficient use of this resource. The bound on buffer requirements on the gateway is given by the combined bandwidth delay product of all connections through the gateway. The maximum combined bandwidth is 1Gbps. The average round trip time is likely to be less then 500ms, giving a worst case buffer requirement of 500Mbits (i.e. 64MB). As network speeds increase, this is likely to increase linearly with the bandwidth as the average round trip time is likely to remain constant (or decrease). The difficult part is reading and writing to this memory at the line speed of the network, but this can be achieved using current technology by having multiple memory banks in parallel and a wide bus to access them.

There are two approaches to initiating a retransmit.

- The TCP on the server node can send the gateway a message to request a retransmit, providing enough detail (the connection ID and sequence number) for the gateway to identify the packet to retransmit. Figure 4(a)
- Alternatively the gateway can monitor incoming traffic, and identify for itself when a retransmit is necessary. Figure 4(b)

The latter clearly simplifies the TCP on the server node, and should result in faster retransmits. This is at the cost of complexity on the gateway as it tends towards a TCP offload engine. The former has the advantage that the gateway need keep no state other than that which is necessary for the DMQs. We are conducting trials to compare these approaches to discover which is more efficient overall. We are effectively researching where the optimal host/network split is for TCP, investigating the compromise between a complete host based implementation and a full TCP offload engine [27].

We have taken the concept of involving the gateway in protocol processing further and started investigating what other roles the gateway can take to assist the server nodes in protocol processing.

A simple, but extremely useful one, is having the gateway strip headers from incoming packets and deliver the headers and payloads to the servers in separate queues, in the same way as is done for intra-cluster communication.

If the gateway is involved in protocol processing and has gone some way to parsing an incoming packet, it could also transmit the parse tree (into a third queue) along with the packet to the server node to prevent the node having to repeat that work. There is again a trade off here between the cluster bandwidth and server node CPU, which we plan to evaluate experimentally.

6. MEASURING PERFORMANCE

We are still at the stage of implementation and optimisation of this approach to networking, so we do not have a complete set of thorough performance measurements. However, the initial experiments do suggest that it will out perform traditional kernel networking (we are currently able to roughly match it).

Bandwidth figures are notoriously difficult to compare, as they are often sensitive to network configuration parameters [18, 2] (such as MTU, TCP window, socket buffer sizes, etc). Published results for other networks also tend to focus on the saturated bandwidth for large packet sizes, which does not give a good measure of the network's performance for real traffic patterns.

We have therefore created our own testbed to allow direct comparison of different technologies. This consists of four PCs acting as the server nodes connected via the CLAN switch. A fifth machine (dual PIII 1GHz STL2) acts as the gateway, and is equipped with a CLAN NIC and a 3Com 985 Gigabit Ethernet NIC. This is connected via Gigabit Ethernet to other machines to allow us to feed traffic to and from the server nodes. This setup is illustrated in Figure 5.

We will be comparing the bandwidth and latency of our approach, both within the cluster and between the cluster and the Internet. This will be compared to a conventional network stack (using Gigabit Ethernet) over the same network topology. The additional latency produced by our cluster and gateway will be measured by comparing path "A" to path "B" in Figure 5. The bandwidth and latency will be measured using *NetPIPE* and *ping*. We will also be using server applications (such as web servers and databases) to gauge how improvements impact the real world.

Finally, we will determine the scalability of our gateway as the number of concurrent connections and the number of nodes within the cluster increases.

7. FUTURE WORK

We are currently at the stage of having a working implementation of the ideas discussed in this paper. This includes a suite of IP based protocols (IP, TCP, UDP, ICMP, BSD sockets) at user level in our CLAN network, and interfacing to other networks via our gateway. We are now beginning a phase of optimisation and performance testing.

This is very much still work in progress, and as such there are some unresolved issues. We are still evaluating (as described in Section 5.8) how much protocol processing on the gateway is possible.

The area of TCP's timers is one that interests us and will be investigated further. In particular, the issues relating to the exact requirements TCP has of timers [3] and the efficient implementation of them at user level [4].

We have yet to implement the extension to preserve received data described in Section 4.2.

There is the possibility of using the gateway and other network hardware to perform load balancing and firewalling operations. The unusual properties of our network (send directed, connection oriented, and the implicit knowledge of other nodes' receive buffer usage) allow for some interesting approaches to these problems.

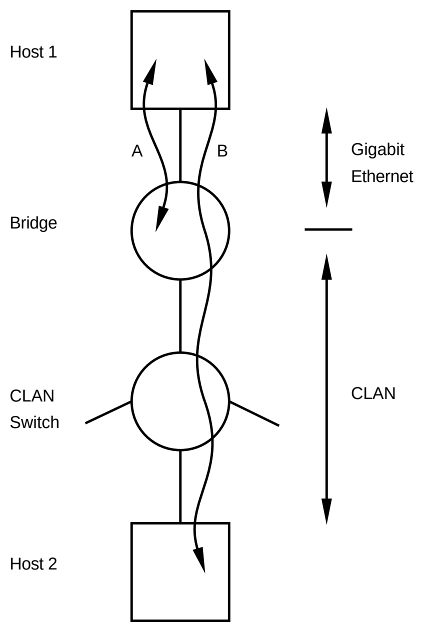


Figure 5: Network Testbed Topology

8. CONCLUSIONS

In this paper we have proposed an innovative approach to networking. By placing the protocols at user level we have been able to achieve high performance, but this benefit comes with a number of problems to which we have suggested solutions.

To make the delivery from network interface to application of incoming packets more efficient, the conventional stack approach has been changed to one where the network interface is a hub for communication, using the traditional stack as a tool to perform protocol processing. This change has resulted in improved performance, but produced its own set of challenges. Zero-copy reception is only possible through modifications to the sockets API, but the extent of these modifications depends on the characteristics of the application. Single-copy operation through an unmodified sockets API is also supported.

True zero-copy transmission of data using the unmodified sockets API has been achieved. This was done by altering the blocking semantics of the sockets operations and moving the role of performing retransmission from the server node to the gateway.

This represents a potentially cheaper and more flexible solution when compared to existing work where co-processors are being used on NICs to perform protocol processing.

9. ACKNOWLEDGEMENTS

The author would like to thank members of the Laboratory for Communication Engineering, Cambridge University Computer Lab, and former members of AT&T Laboratories-Cambridge for their advice. In particular the work done by Steve Pope, Derek Roberts and David Riddoch as part of the CLAN project has been invaluable.

Kieran Mansley is jointly funded by a Royal Commission for the Exhibition of 1851 Industrial Fellowship, and AT&T Labs-Research.

10. REFERENCES

[1] Level 5 Networks Ltd. `http://www.level5networks.com/`.

[2] M. Allman and A. Falk. On the Effective Evaluation of TCP. *ACM Computer Communications Review*, October 1999.

[3] M. Aron and P. Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Dept. of Computer Science, Rice University, 1998.

[4] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Science*, 18(3), August 2000.

[5] S. Bailey and T. Talpey. DDP and RDMA Architecture. Internet Draft `http://www.ietf.org/internet-drafts/draft-ietf-rddp-arch-01.txt`, 2003.

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[8] T. Braun, C. Diot, A. Hoglander, and V. Roca. An Experimental User Level Implementation of TCP. Technical Report 2650, INRIA Sophia Antipolis, France, 1995.

[9] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Operating Systems Design and Implementation*, 1996.

[10] H. K. J. Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference*, 1996.

[11] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

[12] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Communication Protocols. In *Proceedings of ACM SIGCOMM*, 1990.

[13] P. Druschel and G. Banga. Lazy Receive Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[14] A. Dunkels. Minimal TCP/IP Implementation with Proxy Support. Technical Report 20, Swedish Institute of Computer Science (SICS), 2001.

[15] A. Dunkels. Full TCP/IP For 8 Bit Architectures. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, 2003.

[16] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1995.

[17] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *USENIX Annual Technical Conference*, 1999.

[18] P. Gray and A. Betz. Performance evaluation of

copper-based gigabit ethernet interfaces. In *Proceedings of the 27th Conference on Local Computer Networks*, 2002.

[19] P. Howard. VRB - Virtual Ring Buffer. `http://phil.ipal.org/freeware/vrb/`.

[20] IEEE and The Open Group. The Open Group Base Specifications Issue 6. `http://www.opengroup.org/onlinepubs/007904975/basedefs/aio.h.html`.

[21] J.C.Mogul and K.K.Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3), August 1997.

[22] J.Satran, K.Meth, C.Sapuntzakis, M.Chadalapaka, and E.Zeidner. iSCSI. Internet Draft `http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.txt`, 2003.

[23] J. Kay and J. Pasquale. Profiling and Reducing Processing Overheads in TCP. *IEEE/ACM Transactions on Networking*, 4(6), 1996.

[24] D. McAuley and R. Neugebauer. A Case for Virtual Channel Processors. In *Proceedings of the Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, 2003.

[25] S. S. Mukherjee and M. D. Hill. Making Network Interfaces Less Peripheral. *IEEE Computer*, 31(10):70–76, 1998.

[26] I. Pratt and K. Fraser. Arsenic: A User Accessible Gigagit Ethernet Interface. In *Proceedings of Infocom*, 2001.

[27] M. Rangarajan and A. Bohra. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation and Performance. Technical report, Rutgers University, 2002.

[28] D. Riddoch, K. Mansley, and S. Pope. Distributed Computing with the CLAN Network. In *Proceedings of the 27th Conference on Local Computer Networks*, 2002.

[29] D. Riddoch, S. Pope, and K. Mansley. VIA over the CLAN Network. Technical report, University of Cambridge, 2001.

[30] D. Riddoch, S. Pope, D. Roberts, G. Mapp, D. Clarke, D. Ingram, K. Mansley, and A. Hopper. Tripwire: A Synchronisation Primitive for Virtual Memory Mapped Computing. In *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2000.

[31] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local-Area Communication With Fast Sockets. In *USENIX Annual Technical Conference*, 1997.

[32] C. A. F. D. Rose, R. Novaes, T. Ferreto, F. A. D. de Oliveira, M. E. Barreto, R. B. Avila, P. O. A. Navaux, and H.-U. Heiss. The Scalable Coherent Interface (SCI) as an Alternative for Cluster Interconnection.

[33] S.-L.Lo and S.Pope. The Implementation of a Low Call Overhead IIOP-based Object Request Broker. In *Proceedings of ECOOP Workshop on CORBA: Implementation, Use and Evaluation*, 1997.

[34] P. Steenkiste. Design, Implementation, and Evaluation of a Single-copy Protocol Stack. *Software - Practice and Experience*, 28(7):749–772, 1998.

[35] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, Nov 1993.

[36] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net:A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[37] J. Wilkes. Hamlyn—An Interface for Sender-based Communication. Technical Report HPL-OSR-92-13, Hewlett-Packard Research Laboratory, November 1992.