A Low Overhead Application/Device-driver Interface for User-level Networking

D. Riddoch
Laboratory for Communications Engineering
Department of Engineering
University of Cambridge, England

S. Pope
AT&T Laboratories Cambridge
24a Trumpington Street
Cambridge, England

Abstract

Recent user-level network interfaces have placed an increasing proportion of their functionality in hardware, in order to provide an efficient user-level interface. The performance of such systems is significantly better than that of traditional network architectures, but comes at a cost; namely increasing complexity of the hardware, reduced flexibility and limited scalability.

In this paper we present a technique that reduces the overhead of the application/device-driver interface, hence shifting the tradeoff towards a soft implementation. We show how this technique is used to improve the performance of the CLAN user-level network interface. Results given show that the performance of this interface exceeds that of other technologies that provide a direct user-level interface to the hardware, whilst retaining the flexibility and simplicity of a software interface.

Keywords: user-level, networking, device-driver, asynchronous

1 Introduction

The performance of traditional network architectures is largely limited by the high software overhead on the host[1]. Contributions include the cost of making system calls, demultiplexing, protocol stacks and copying of data between address-spaces and buffers. In the last few years a number of projects have addressed these problems by providing a user-level interface to the network interface[2, 3]. Applications access the network directly through a virtual memory mapping onto the hardware and/or through a region of host memory shared with it. In some cases the network provides reliable data delivery, allowing very simple, low-overhead protocols, since there is no need for error detection, correction or retransmission. The reduced software overhead also significantly improves cache performance.

These innovations have contributed to greatly improved throughput and latency, and reduced overhead. However, they come at a price; namely increasing complexity (and hence cost) of the hardware, and reduced flexibility and scalability. If a resource is to be accessible at user-level by multiple applications, there need to be multiple independent instances of that resource in hardware. The number of instances limits the number of applications or endpoints than can be supported simultaneously. An example is the Virtual Interface Architecture (VIA)[4], where significant hardware resources are needed for each endpoint[5].

Thus there is a tradeoff inherent in deciding whether a particular feature should have a user-level accessible interface, or be managed by the device-driver. Existing systems strike a balance by attempting to provide those features that are on the critical path at user-level. Typically data transfer can be achieved entirely at user-level, whereas connection management and synchronisation require a system call.

In this paper we describe an interface between the application and device-driver that reduces the overhead incurred when using resources that are managed by the device-driver. A region of memory shared by the device-driver and application is used to pass various data and control messages asynchronously. Applications pass commands to the device-driver, and receive messages and other out-of-band data entirely at user-level, significantly reducing the number of system calls.

2 The CLAN Network Interface

As part of the CLAN[6] project at AT&T Laboratories Cambridge we have developed a gigabit class user-level network interface controller (NIC). The basic communications model is non-coherent distributed shared memory, wherein a portion of the virtual address space of an application is logically mapped over the network onto physical memory in another node. This is implemented by a virtual address mapping onto the NIC, which forwards read and write requests over the network, as illustrated in Figure 1.

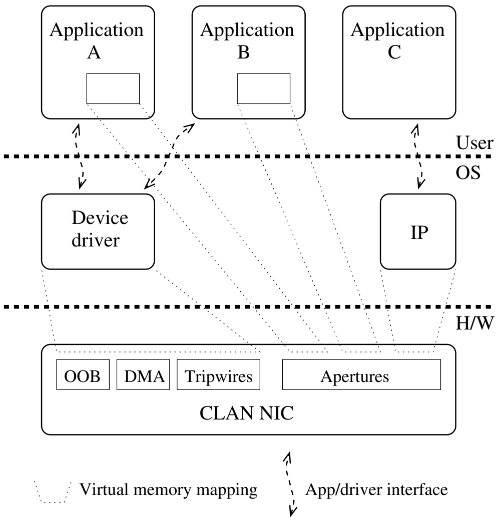

Figure 1: The CLAN network interface.

An application may set up an mapping onto memory in an application on a remote node, known as an *aperture*. Communication is achieved by writing (and reading) to (from) the aperture using programmed I/O or DMA.

As shown in the figure, the NIC provides a number of additional resources, which are only directly accessible by the device driver:

Fixed size *out-of-band messages* are sent between endpoints, and are used for connection management and miscellaneous control. Each endpoint has a (configurable) fixed length queue for out-of-band messages, and when this is full further messages are discarded. Message reception is driven by an interrupt from the NIC.

The NIC has a programmable remote direct memory access (RDMA) engine, which transfers blocks of data from local memory to a remote location. The transfer is performed in parallel with processing on the CPU, thus providing low overhead data transfer for medium and large messages. This shared resource is multiplexed by the device-driver.

Distributed shared memory itself provides no efficient means to synchronise with the arrival of data from the network, since it is delivered into an application's address space asynchronously. The CLAN NIC provides a novel primitive, the *tripwire*, which provides efficient and flexible synchronisation with cross-network accesses to arbitrary shared memory locations. A tripwire is associated with some memory location, and *fires* when that location is read or written via the network. The CLAN network and tripwire synchronisation primitive are described in detail elsewhere[7].

3 Application/Driver Interface

The main technique used to pass information between the application and device-driver is the *asynchronous queue*. This is simply a circular buffer, similar to those commonly used within applications, but placed in a segment of memory that is mapped into the address-space of both the application and device-driver.

Two pointers (`read_i` and `write_i`) give the positions in the buffer of the head and tail of the queue, the consumer incrementing the read pointer modulo the size of the queue, the producer managing the write pointer likewise. The traditional implementation uses an additional flag to distinguish the full and empty cases, and requires mutual exclusion to ensure consistency.

However mutual exclusion between the operating system (OS) and application is difficult to achieve without using system calls. Instead we eliminate the flag, and define the queue to be empty when the pointers are equal, and full when there is a single free slot in the buffer. Atomic integer updates ensure consistency.

3.1 Out-of-band message queues

An asynchronous queue is used to pass out-of-band messages from the device-driver to CLAN communication endpoints. It is essential that the OS be isolated from the (mis)behaviour of the application, and so must not rely on the validity of data in the memory segment shared with the application. Thus the device-driver keeps its description of the layout and state of the queue in private memory that is not shared with the application, as illustrated in Figure 2.

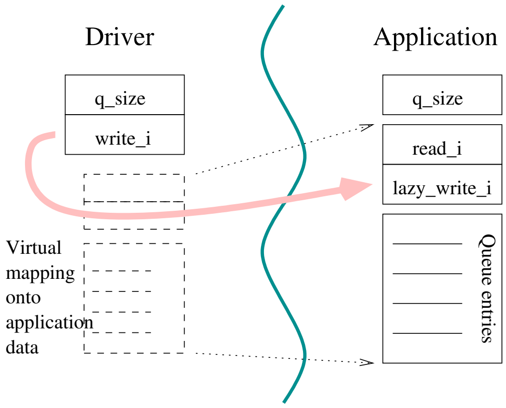

Figure 2: Asynchronous message queue.

The application needs to be able to *read* the value of the write pointer, so a copy (`lazy_write_i`) is maintained in the shared region, and is updated whenever `write_i` is updated. The C code to enqueue a message is given in Example 1. If the queue fills, further messages are discarded.

Example 1 Out-of-band message delivery.

```
void oob_q_put(const oob_msg* msg)
{
    next_write_i = (write_i + 1) % q_size;
    if( next_write_i == shm->read_i )
        /* discard */ return;
    shm->q[write_i] = *msg;
    MEMORY__BARRIER();
    write_i = next_write_i;
    shm->lazy_write_i = write_i;
    if( any_processes_blocked )
        wake_them_up();
}
```

As shown in Example 2, the application dequeues messages entirely at user-level; a system call is only required if the application wishes to block waiting for a message. Polling for messages is thus very cheap, which is critically important for a user-level network, because applications must frequently check for messages when communicating in order to determine if the remote endpoint has been closed. This is done on the receive side whenever there is no data immediately available, and on the transmit side before sending data.

Example 2 Out-of-band message retrieval.

```
int oob_q_get(oob_msg* msg, int blocking)
{
    if( read_i == lazy_write_i )
        if( blocking )  oob_q_wait();
        else  return Q_WOULDBLOCK;
    *msg = q[read_i];
    MEMORY__BARRIER();
    read_i = (read_i + 1) % q_size;
    return Q_OK;
}
```

3.2 RDMA request queues

A similar queue is used to pass RDMA requests in the other direction; from the application to the device-driver. A request consists of a pointer to the source in local memory, a descriptor for the destination in remote memory, and the length. The device-driver maintains a list of RDMA request queues that are currently active, and services them in a round-robin fashion. Requests are dequeued, checked for correctness and then passed on to the NIC. When a request completes an interrupt is generated, and the interrupt-service-routine is used to start the next request. Where possible, a request is checked and mapped in parallel with the execution of the previous request, in order to minimise the turn-around time between requests.

When the first request is placed in the queue, the application makes a system call to let the device-driver know that the queue is active, but after that the device-driver dequeues entries asynchronously. The application may continue to add entries to the queue, and only need invoke the device-driver again if the queue empties completely.

By multiplexing multiple soft queues onto a sin-

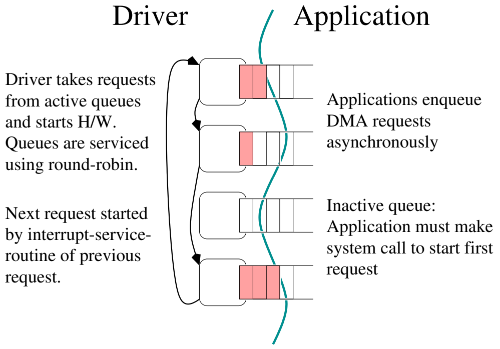

Figure 3: RDMA Request Queues

gle hardware resource (rather than providing multiple hardware queues accessible from user-level) we can be flexible about scheduling requests. Our implementation allows applications to indicate that a number of RDMA requests form part of a single logical message, and so should be scheduled together. It would also be trivial to implement a priority scheme or set a maximum transfer size and split large requests in order to prevent a single application from monopolising the resource.

3.3 Tripwire notification

Applications have to make system calls to initialise, enable and disable tripwires. These are typically done at connection setup or when the application needs to block, and so are not on the critical path. The state of a tripwire — whether it has fired or not — is made available at user-level in a bitmap stored in memory shared with the device-driver, and thus can be queried with very low overhead. Large numbers of tripwires can be polled efficiently be examining a whole word at a time.

3.4 Asynchronous event queues

The CLAN NIC generates a variety of events, including tripwires, RDMA completion and out-of-band message arrival. A unified mechanism and interface is needed to deliver these events from multiple endpoints, and allow efficient synchronisation with event arrival. Traditionally some variant on the BSD *select* system call is used. However, the deficiencies of this style of interface are well known, and even with sophisticated optimisations[8] have been shown to incur high overhead and scale poorly with the number of end-points.

We have developed a mechanism which delivers events directly to the application with low overhead, and with cost $O(1)$, using the asynchronous queue described above. The queue resides in memory mapped into the OS and application's address space, and so the application can dequeue events entirely at used-level, and only need make a system call in order to (un)register events and to block.

We have generalised the Linux `wait_queue` by adding a call-back,[1] which is used here to enqueue an event. To prevent the queue from overflowing we ensure that each registered event can only be enqueued once before it is dequeued. This is implemented by a bitmap which has a single bit per registered event, and which also lives in the shared memory segment.

The wait-queue call-backs provide a generic event hook which we have also used to implement an in-kernel event-driven NFS service, and a performance improvement for the *select* and *poll* system calls.

4 Performance

In this section we present a number of micro benchmarks that characterise the performance of the CLAN NIC and software interface. For comparison we also give results for the Giganet cLAN 1000[9] implementation of VIA, which presents request queues and completion events as hardware resources that are directly accessible at user-level.

4.1 The test platform

The benchmarks were run on a pair Pentium III workstations running the Linux 2.2 operating system. The CLAN NICs were connected via a switch, the Giganet NICs connected back-to-back. The free-running cycle counter was used for fine-grained performance measurements.

1. The existing implementation can only be used to wake a sleeping process

The test application consists of a server and client. The server is single-threaded, with all completion events multiplexed onto a single asynchronous event queue for CLAN, and a completion queue for VIA. The client application makes one or more connections to the server and sends small messages, which are acknowledged. After processing each request the server spins for up to $15\mu s$ before blocking, in order to improve response and reduce overhead when the server is under load[10].

4.2 Results

Figure 4 shows the bandwidth achieved when streaming messages of various sizes. Flow control is provided by acknowledgements sent from the receiver to the sender, which also indicate the amount of buffer space available. The DMA performance of the CLAN NIC is comparable with the performance of VIA, and in both cases the small message bandwidth appears to be limited by the inter-request setup time.

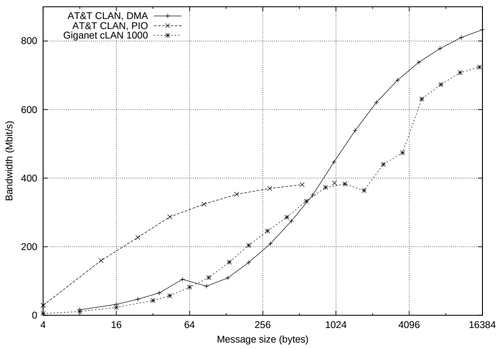

Figure 4: Bandwidth vs. message size.

On the CLAN NIC this can be alleviated by using programmed I/O, which for small messages has low overhead and consequently superior throughput. The maximum throughput here is limited by the memory system on the test nodes. We have achieved over 900 Mbps on Alpha workstations.

Figure 5 shows the round-trip time for a small message varies with the number of idle connections that the server is managing. The plot shows that for both CLAN and VIA the round-trip time scales well with respect to the number of connections. The two plots (for each interface) show the round-trip time (a) when the server is idle and so has to rescheduled when a message arrives, and (b) when the server is kept busy, and so already running on the processor when the message arrives.

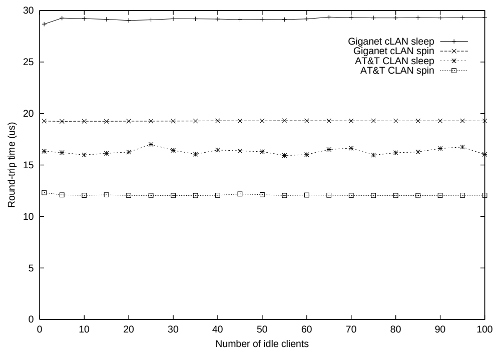

Figure 5: Round-trip time vs. idle endpoints.

The final test attempts to show how the interface performs under load. Because we only have two VIA test nodes, a single client application is used to make multiple connections to the server, and simulates the behaviour of many independent clients. An increasing number of connections are opened to the server, and a request is sent down each connection. The results are shown in figure 6.

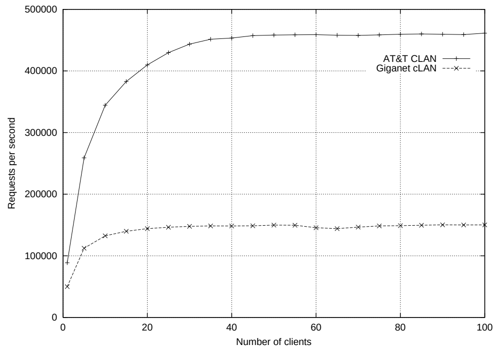

Figure 6: Throughput vs. offered load.

For the VIA result it is not clear whether the bottleneck is in the client or the server, since they have to perform similar amounts of work. However, the total per-message overhead can be estimated, and is $2.2\mu s$ for CLAN and $6.7\mu s$ for VIA.

5 Related Work

The Piglet operating system is an attempt to bring enhanced performance and quality of service to multi-processor systems by partitioning the functionality of the OS between processors. Processors in the system can be associated with I/O devices, and run a lightweight device kernel (LDK) which provides the interface for the device, and may implement some of the functionality of the networking subsystem. The LDK communicates with the host OS and also directly with applications via a shared memory interface[11]. Endpoints are polled in order to provide low latency, but the effect on scalability has not been explored.

In the Nemesis[12] operating system I/O channels used to transfer data between processes are implemented by passing buffer descriptors (*iorecs*) through an asynchronous queue. Event counts give the position of the head and tail of the queue, and also provide synchronisation. The event counts are an operating system primitive, and are updated by a system call.

A number of solutions to the problem of delivering I/O events from multiple sources have been proposed, including POSIX realtime signals, the `/dev/poll` interface in Solaris and others[13]. The main contribution of the asynchronous event queue given in this paper is the ability to dequeue events entirely at user-level, and hence with very low overhead.

6 Conclusions

The CLAN network provides a distributed shared memory based communications model, with a additional facilities for connection management, synchronisation and DMA. Although these key aspects of the network interface are managed by the device-driver, use of a carefully designed shared memory interface leads to very low overhead. All the facilities needed on the critical path of communication can be accessed at user-level.

The performance of the RDMA engine is comparable with that of the Giganet cLAN VIA NIC, but because it is managed by the device-driver, requires considerably less hardware support. Supporting increasing numbers of applications and endpoints only requires more memory, whereas for VIA additional resources are also needed in the NIC. Further, support for programmed I/O in the CLAN NIC provides very low overhead and latency for small messages.

The RDMA engine currently requires an interrupt per transfer, which adds significant overhead and damages applications' cache performance. A future revision of the NIC will allow multiple requests to be submitted at once, and will thus generate fewer interrupts. This will also reduce the turn-around time between requests, and hence improve small message bandwidth.

The current trend in high performance networking is to make more of the network interface accessible at user-level by providing increasing amounts of functionality in hardware. However, the CLAN network succeeds in delivering high bandwidth with low latency and overhead with an implementation that is simpler and more scalable. By reducing the overhead of the application/device-driver interface, the tradeoff shifts away from a user-level accessible hardware implementation and towards device-driver management, with the associated benefits for scalability, flexibility and cost.

Acknowledgements

The authors would like to thank all of the members of the Laboratory for Communications Engineering, and AT&T Laboratories Cambridge, particularly members of the CLAN team.

David Riddoch is jointly funded by the Royal Commission for the Exhibition of 1851 and AT&T Laboratories Cambridge.

References

[1] Richard Martin, Amin Vahdat, David Culler, and Thomas Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *24th International Symposium on Computer Architecture*, 1997.

[2] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-

Level Network Interface for Parallel and Distributed Computing. In *15th ACM Symposium on Operating Systems Principles*, December 1995.

[3] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] *The Virtual Interface Architecture*. `http://www.viarch.org/`.

[5] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Supercomputing*, November 1998.

[6] The CLAN Project. `http://www.uk.research.att.com/clan/`.

[7] David Riddoch, Steve Pope, Derek Roberts, Glenford Mapp, David Clarke, David Ingram, Kieran Mansley, and Andy Hopper. Tripwire: A Synchronisation Primitive for Virtual Memory Mapped Communication. In *4th International Conference on Algorithms and Architectures for Parallel Processing*, December 2000.

[8] Gaurav Banga and Jeffrey Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX Technical Conference*, June 1998.

[9] Giganet, Inc. `http://www.giganet.com/`.

[10] Stefanos Damianakis, Yuqun Chen, and Edward Felten. Reducing Waiting Costs in User-Level Communication. In *11th International Parallel Processing Symposium*, April 1997.

[11] Steve Muir and Jonathan Swift. Functional divisions in the Piglet multiprocessor operating system. In *ACM SIGOPS European Workshop*, 1998.

[12] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[13] Gaurav Banga, Jeffrey Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Technical Conference*, June 1999.