Distributed Computing with the CLAN Network

David Riddoch
Department of Engineering
University of Cambridge
Cambridge, England
djr23@cam.ac.uk

Kieran Mansley
Department of Engineering
University of Cambridge
Cambridge, England
kjm25@cam.ac.uk

Steve Pope
AT&T Laboratories-Cambridge
24a Trumpington Street
Cambridge, England
steve.pope@cl.cam.ac.uk

Abstract

CLAN (Collapsed LAN) is a high performance user-level network targeted at the server room. It presents a simple low-level interface to applications: connection-oriented non-coherent shared memory for data transfer, and Tripwire, a user-level programmable CAM for synchronisation. This simple interface is implemented using only hardware state machines on the NIC, yet is flexible enough to support many different applications and communications paradigms.

We show how CLAN is used to support a number of standard transports and middleware: MPI, VIA, TCP/IP and CORBA. In each case we demonstrate performance that approaches the underlying network. For TCP/IP we present our initial results using an in-kernel stack, and describe the architecture of our prototype Gigabit Ethernet/CLAN bridge, which demultiplexes Ethernet frames directly to user-level TCP/IP stacks via the CLAN network. For VIA we present a software implementation with better latency than a commercial VIA NIC implemented on ASIC technology.

Keywords: CLAN, high performance, user-level networks, network interface.

1. Introduction

As the line speed of local area networks reaches a gigabit per second and beyond, the overhead of software on the host system is increasingly becoming the limiting factor for performance. At high message rates the processing time is dominated by network overheads, at the expense of the application, and can lead to performance collapse.

The overhead is due to a number of factors[11] including copying data between buffers, protocol processing, demultiplexing, interrupts and system calls. In addition to the processor time taken, these activities have a detrimental effect on the cache performance of the application.

One solution that addresses these problems is *user-level networking*, wherein applications communicate directly with the network interface controller (NIC), bypassing the operating system altogether in the common case. The NIC typically has direct access to application buffers, eliminating unnecessary copies. In some cases the network provides a reliable transport, which simplifies protocol processing.

A variety of user-level network interfaces have been developed[28, 8, 9], each supporting a particular communications paradigm. For example, SCI has largely been used to support shared-memory scientific clusters, and Arsenic[24] supports processing of TCP and UDP streams. Other communication interfaces can be built as layers of software above the raw network, but this typically incurs significant additional overhead when the two interfaces are dissimilar.

One approach to supporting multiple network interfaces is to use a programmable NIC. Myrinet[9] is a gigabit class user-level accessible NIC which incorporates a processor. A number of communications interfaces have been built using Myrinet, including MPI[25], the Virtual Interface Architecture (VIA)[7, 10], VMMC-2[13] and TCP/IP[15]. However, at any one time all communicating nodes must be programmed to support the same model.

The CLAN network presents a single, low-level network interface that supports communication with low overhead and latency, high bandwidth, and efficient and flexible synchronisation. In this paper we show how this interface supports a range of disparate styles of communication, without sacrificing the performance of the raw network.

MPI, VIA and CORBA are implemented as user-level libraries, requiring no privileged code or modifications to the network. We present an in-kernel IP implementation, and also describe the architecture of our Gigabit Ethernet/CLAN bridge, which demultiplexes Ethernet frames directly onto user-level TCP/IP stacks via the CLAN network.

2. The CLAN Network

CLAN is a high performance user-level network designed for the server room. Key aims of the project include support for general purpose multiprogrammed distributed systems, and scalability to large numbers of applications and endpoints. An overview of the key features of the network follows:

At the lowest level the communications model is non-coherent distributed shared memory (DSM). A portion

of the virtual address space of an application is logically mapped over the network onto physical memory in another node. Data is transferred between applications by writing to the shared memory region using standard processor write instructions. A buffer in a remote node is represented by an Remote Direct Memory Access (RDMA) cookie, the possession of which implies permission to access that buffer.

However, the CLAN network is not intended to support the traditional DSM communications model. Instead, the shared memory interface is used as the low-level data transfer layer on which higher-level communications abstractions are built. The network supports small datagram messages, which are currently used for connection management. The NIC also provides a programmable DMA engine to off-load data transfer from the CPU.

2.1. Simple data transfer

By way of example, we present the implementation of a simple message passing protocol. The Distributed Message Queue is based on a circular buffer in memory local to the receiver, as illustrated in Figure 1. The sender writes a message through its mapping onto the receive buffer, at the position indicated by the write pointer (`write_i`). The write pointer is then incremented modulo the size of the buffer, and the new value copied to the remote address-space (`lazy_write_i`).

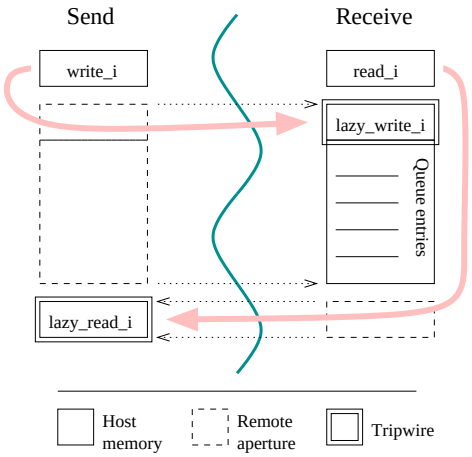

Figure 1. A Distributed Message Queue

The receiver compares its lazy copy of the write pointer with the read pointer to determine whether or not the queue is empty. Messages are dequeued by reading them from the buffer, then incrementing the read pointer, and copying its new value to the sender.

Transferring small messages in this way consists of just a few processor write instructions, and hence has very low overhead.

2.2. RDMA cookie-based communication

In some cases, it is possible to arrange for the application to read received data directly from in the circular buffer (*in-place*). Other programming interfaces require data to be delivered to application-level receive buffers, which requires an additional copy.

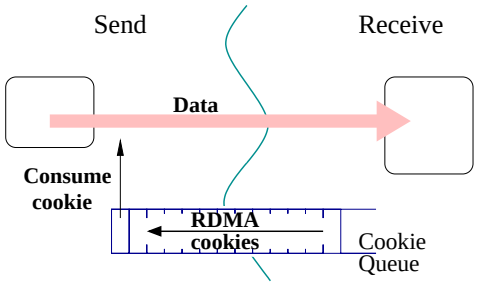

Figure 2. RDMA cookie-based data transfer.

This copy can be avoided if the sender is informed of the location of the receive buffers in advance. To achieve this, the receiver sends RDMA cookies for its buffers to the sender using a distributed message queue, known as a *cookie queue*. The sender retrieves an RDMA cookie from the cookie queue, and uses it as the target for a DMA transfer. This is illustrated in Figure 2.

2.3. Synchronisation

On the receive path, data is placed in an application's buffers asynchronously, without the intervention of the CPU. This minimises overhead, but means the application has no way to determine that a message has arrived other than by polling memory locations explicitly.

Other networks have solved this problem in one of two ways: by being able to request that an interrupt be generated when a particular region of shared memory is accessed, or by using some form of out-of-band synchronisation messages.

The CLAN NIC provides a novel solution: the *tripwire*[27]. A tripwire is an entry in a content addressable memory (CAM) which matches a particular address in an application's address space. The address of each memory location that is accessed via the network is looked-up in the CAM, and when there is a match the application receives a notification. If the application is blocked waiting for such a notification, an interrupt is generated and the application is rescheduled.

Tripwires are programmed directly by user-level applications, and are set on locations that correspond to protocol specific events. For example, when using the distributed message queue above, the receiver sets a tripwire on `lazy_write_i`, and receives a notification whenever a new message is placed in the queue. If the receiver is blocked waiting for a new message, it will be rescheduled. Similarly the sender can block waiting for space in the queue by setting a tripwire on `lazy_read_i`.

This is a flexible and fine-grained solution to the synchronisation problem. With tripwires, synchronisation is orthogonal to data transfer, and decoupled from the transmitter. This greatly simplifies the hardware implementation. A tripwire can be associated with control signals as above, or alternatively with in-band data.

2.4. Event handling

The NIC generates a variety of events, including DMA completion, out-of-band message arrival and tripwire events. Any of these can be directed to an *asynchronous event queue*[26]. This is a shared memory data structure, which allows events to be dequeued at user-level with very low overhead. Once an event has been enqueued it is blocked, so the queue is not susceptible to overflow. The CPU overhead of event delivery is $O(1)$ with respect to the number of events registered with the queue.

2.5. Prototype implementation

The prototype CLAN NICs are based on off-the-shelf parts, including an Altera 10k50e FPGA clocked at 60 MHz, a V3 PCI bridge (32 bit, 33 MHz) and HP's G-Link optical transceivers with 1.5 Gbit/s link speed. We have also built a five port worm-hole-routed switch, again using FPGAs, and a non-blocking crossbar switch fabric. A bridge to Gigabit Ethernet is at the debug stage.

The tripwire synchronisation primitive is implemented by a content addressable memory, supporting 4096 tripwires in the current version. Tripwires are managed by the device driver, and when an tripwire fires an interrupt is generated. The interrupt service routine delivers an event to the application, and wakes any processes waiting for the event.

The V3 PCI bridge chip includes an integrated DMA engine, which can only be programmed with a single request at a time, and generates an interrupt after each transfer. The interrupt service routine then starts the next DMA request. This causes a large gap between each DMA request, which severely limits DMA performance for small and medium sized messages.

The format of data packets on the wire resembles that of write bursts on a memory bus. The header identifies the target node and address of the first word of data. The amount of data in the packet is not encoded in the header, but is implicit in the data which follows. The packet can thus be split at any point, and a new header generated for the trailing portion. Conversely, consecutive packets that represent a contiguous transfer can be merged into a single packet by a switch or receiving NIC.

Because the packet length is not encoded in the header, the NICs and switches can begin to emit packets as soon as data is available, rather than waiting for an entire packet. This contributes to the low latency of the CLAN network. The switch exploits the ability to split packets to prevent large packets from hogging an output port unfairly. No maximum packet size is enforced, so the network operates as efficiently as the traffic patterns allow. If congestion is encountered, small packets are likely to be merged, leading to larger packets and higher efficiency.

Flow control is rate-based on a per-hop basis, with flow control information passed in-band with the data. This ensures that the source rate can be adjusted in a timely fashion to prevent buffer overruns in the receiver. The NICs and each switch port have just 512 bytes of buffer space.

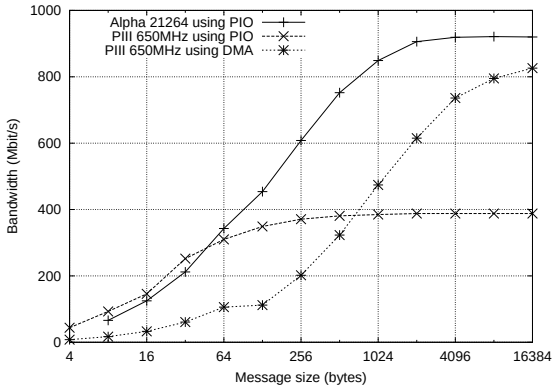

Figure 3. Raw bandwidth vs. message size.

2.6. Scalability

The data path in the CLAN NICs and switches is simple compared with other technologies which run at the same line speed. It is implemented entirely as hardware combinatorials and state machines, and runs at full speed on three year old FPGA technology. We have recently completed a design to run at 3 Gbit/s, also using an FPGA. These factors indicate that the network model is likely to scale to significantly higher line speeds, if necessary with integration.

The maximum number of endpoints that can be supported in a user-level network is usually limited by per-endpoint resource in the NIC. In CLAN NICs, per-endpoint resource just consists of incoming and outgoing aperture mappings and tripwires, so a large number of endpoints can be supported relatively cheaply.

3. Baseline Performance

3.1. Test configuration

The test system consisted of a pair of off-the-shelf PC systems connected through a CLAN switch. Each node was a 650 MHz Intel Pentium III system with 256 MB SDRAM and 256 KB cache, running an unmodified Linux 2.4.6 kernel. Except where otherwise stated, all performance results given in this paper were measured using this configuration. The error in the graphs is too small to represent with error bars.

3.2. Latency and bandwidth

Application to application latency for single word programmed I/O (PIO) writes was measured by timing a large number of ping-pongs (100000). The median round-trip time was 5.6 µs, with 98.6% below 5.7 µs. Measurement with a logic analyser showed that the switch contributed .8 µs in each direction.

The bandwidth was measured by streaming a large amount of data through the distributed message queue described in Section 2.1, using a 50 KB buffer. The results

are shown in Figure 3. All data is touched on both the transmit and receive side.

For small messages DMA performance is limited by the V3 bridge's DMA engine – which has high overhead and a high turn-around time between requests. The kink between 64 and 128 bytes is due to an optimisation in the DMA driver, where PIO is used for small messages.

PIO gives excellent performance with low overhead for small messages, but is limited by the PC I/O system to less than 400 Mbit/s. Using an Alpha 21264 system are we able to saturate the network, achieving up to 960 Mbit/s with PIO, and half bandwidth is available with messages of just 100 bytes. An improved DMA engine with a user-level interface and pre-fetching ought to achieve performance that is much closer to this curve.

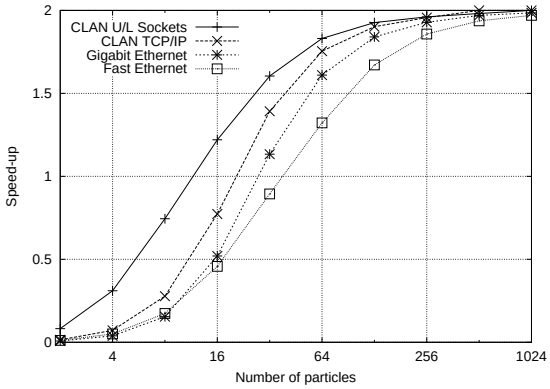

Figure 5. MPI n-body calculation speed-up using two nodes.

4. MPI

MPI is the defacto-standard communications interface for parallel scientific computing, and is widely implemented and used. It has been designed to be efficient on a variety of architectures, from shared-memory multiprocessors to networks of workstations. The interface is based on message passing, and includes primitives for both point-to-point communications and a variety of collective operations, including multicast.

4.1. Implementation

Our port of MPI is based on the LAM[3] implementation, which runs over the standard BSD socket interface. All collective operations are implemented in terms of point-to-point connections. We have replaced the standard socket calls with a user-level socket library that provides the same semantics. Our socket library is based on the distributed message queue described in Section 2.1.

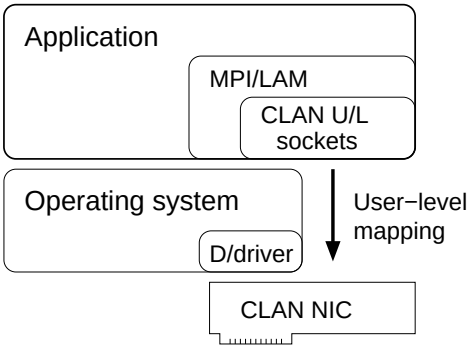

Figure 4. The architecture of CLAN MPI.

The round-trip time for small messages using `MPI_Send()` and `MPI_Recv()` is 15 µs. This compares with 19 µs for MPI-BIP[25] using Myrinet hardware, and 33 µs for MPI over FM over the Emulex cLAN 1000[21].

To demonstrate a real application, we chose a standard n-body problem. It is representative of the applications that can be solved with loosely coupled networks of processors, yet is not perfectly parallelisable (so is a good indicator of network performance).

Figure 5 shows the speed-up achieved using two nodes. We compare MPI over Fast Ethernet, Gigabit Ethernet (3c985), CLAN kernel-level IP (see Section 6.1) and CLAN MPI. This problem is latency constrained for small numbers of particles, so MPI over CLAN at user-level does substantially better than MPI over TCP/IP. CLAN MPI has very low overhead, so will also give improved performance to applications that are not sensitive to latency.

5. Virtual Interface Architecture

The Virtual Interface Architecture is an industry standard[6] for user-level networking. Its scope is to describe an interface between the NIC and software on the host, and an application programming interface[2]. The intention is that vendors develop and market devices that implement this specification, such as Emulex's cLAN 1000[1].

Alternatively, VIA can be provided on existing networks by emulating the API in software. M-VIA[5] consists of a user-level library and loadable kernel module for Linux, and supports VIA over Ethernet. A third approach is to use an intelligent NIC. Intel's proof-of-concept[7] implementation and Berkeley VIA[10] both use Myrinet.

5.1. VIA data transfer

In the standard send/receive data transfer model, a sending process enqueues descriptors for source buffers by calling `VipPostSend()`, which returns immediately. The send operation completes asynchronously, and the application can poll for completion by calling `VipSendDone()`, or block waiting for completion with `VipSendWait()`.

Similarly the receiving process posts descriptors describing buffers to the receive queue using `VipPostRecv()`. These descriptors are completed when data is delivered into the buffers, and the ap-

plication synchronises using `VipRecvDone()` and `VipRecvWait()`.

To support applications that manage multiple connections, notifications of completed requests from a number of VIA endpoints can be directed to a *completion queue*. The application can poll the completion queue (`VipCQDone()`), or block waiting for events (`VipCQWait()`). The returned value indicates which endpoint the descriptor completed on, and whether it was a send or receive event.

5.2. Implementation

We have implemented the VIA API as a user-space software library over CLAN. The architecture is similar to that of our MPI implementation, shown in Figure 4. The functionality we have so far includes the send/receive data transfer model, polling and blocking modes of synchronisation, completion queues and all three reliability levels. This is sufficient to provide source-level compatability for many VIA applications.

5.2.1 Data transfer. Basic data transfer is illustrated in Figure 6. The receiving application posts a receive descriptor to an endpoint (1) using `VipPostRecv()`. The segments within the descriptor are mapped to CLAN RDMA cookies, and passed to the remote endpoint via a cookie queue, as described in Section 2.2. Control is returned to the application immediately.

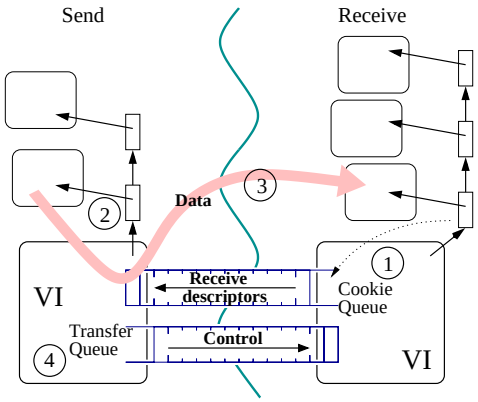

Figure 6. VIA data transfer.

Some time later (2), the sending application posts a send descriptor. The cookie queue is interrogated to find the RDMA cookies for the receive buffers, and one or more DMA requests are made to transfer the application data directly from the send buffers to the receive buffers (3). A second message queue, the *transfer queue*, is used to pass meta-data (including the size of the message) and control from the sender to the receiver (4).

Data transfer itself happens asynchronously when the DMA requests reach the front of the DMA queue. Alternatively, the data can be transferred by PIO, which has lower overhead and latency for small messages. This requires a memory mapping onto the remote receive buffer, which is relatively expensive to set up, and so a cache of such mappings is maintained.

A further benefit of PIO for small messages is that data transfer happens during the application's scheduling time slice, rather than when the *NIC* chooses to schedule the transfer, as for DMA. Applications that have delay sensitive traffic can use PIO to ensure timely delivery of messages, even when competing with large transfers. Thus the operating system's process scheduling policy also manages network access. Jitter introduced by the network is very small (at most 20.5 µs per hop), so good quality of service can be achieved with a real-time scheduler.

5.2.2 Synchronisation. Send synchronisation is trivial, and merely involves determining whether the DMA requests associated with a descriptor have completed. This information is provided by the CLAN DMA interface.

The completion of an incoming message is indicated by the arrival of a message in the transfer queue. For the non-blocking `VipRecvDone()` method this can be detected by inspecting the transfer queue. To support blocking receives (`VipRecvWait()`) a tripwire on the transfer queue is used as described in Section 2.3.

The VIA completion queue is implemented using the CLAN asynchronous event queue. When an endpoint's receive queue is associated with a completion queue, a tripwire is attached to the transfer queue, and configured to deliver events to the event queue. To associate an endpoint's send queue with a completion queue, DMA completion events are directed to the event queue.

5.2.3 Flow control. If the cookie queue is found to be empty when a send descriptor is posted, then the receive buffers have been overrun, and VIA specifies that the data should be dropped. This condition is detected without any data being transmitted across the network, so the network is not loaded with data that cannot be delivered.

To avoid packet loss, applications have to build flow control on top of VIA. To get good performance, flow control information must be timely, and this cannot be achieved if it is being multiplexed over the same channel as bulk data. To address this, Emulex VIA provides non-standard interfaces for communicating out-of-band information with low latency.

In our implementation, an application may configure an endpoint to queue-up send descriptors until corresponding receive buffers are posted. This is possible because the sending application receives a notification when a message is placed in the cookie queue. This extension to the standard improves performance, simplifies application code considerably, and requires no additional non-standard primitives.

5.2.4 Protection. Due to lack of space on the FPGA, our prototype NIC hardware currently lacks full protection on the receive path. Having given a remote process access to a buffer it is not possible to revoke access. This means that a faulty or malicious node that goes in below the level of VIA can overwrite data in an application's receive buffers after the receive descriptor has completed, which could cause the application to misbehave. Proper

protection will be available in a future revision of the NIC.

5.3. Performance

In this section we compare the performance of our implementation of VIA with that of an existing commercial implementation: the Emulex cLAN 1000. The Emulex NIC is a 64 bit, 33 MHz PCI card, with a single chip implementation and 1.25 Gbit/s link speed. We did not have access to an Emulex switch for these tests, so the Emulex NICs were connected back-to-back. The system setup and benchmark programs for the two systems were identical.

Since many distributed applications require reliable communications, the reliability level used in these tests was *reliable delivery*. To prevent receive buffer overrun, the test applications used credit-based flow control. For CLAN VIA we present separate results for PIO and DMA data transfer for clarity (although by default we switch between the two dynamically).

The latency for small messages was measured by timing a large number of round-trips. This value includes the time taken to post a send descriptor, process that descriptor, transfer the data, synchronise with completion on the receive side and make the return trip. The results are given in Table 1.

Table 1. Round-trip time for VIA (µs).

Bytes transferred	CLAN (DMA)	CLAN (PIO)	Emulex cLAN 1000
0	10.7	8.5	12.6
4	14.5	11.6	14.5
40	15.5	12.1	18.3

The small message latency for CLAN VIA (PIO) is the lowest by some margin, despite the fact that the CLAN NICs are connected by a switch, whereas the Emulex NICs are connected back-to-back. Without a switch, the CLAN VIA (PIO) round-trip time is just 6.7 µs. For comparison, M-VIA report latency over Gigabit Ethernet of 38 µs[4], and Berkeley VIA over Myrinet report 46 µs[10].

The bandwidth achieved for various message sizes is given in Figure 7. Data is 'touched' on both the send and receive side. For messages up to 128 bytes, CLAN VIA (PIO) has the highest throughput. CLAN VIA (DMA) performs poorly for small messages due to the high overhead of the V3's DMA engine. The kink between 64 and 128 bytes is due the DMA optimisation described in Section 3.2.

We have also measured maximum transaction rates with a server application that simply acknowledges each message it receives. With about 15 clients Emulex VIA saturates at 150,000 requests per second. The same application implemented over the *raw* CLAN network is able to process 1,030,000 requests per second – a seven-fold improvement.

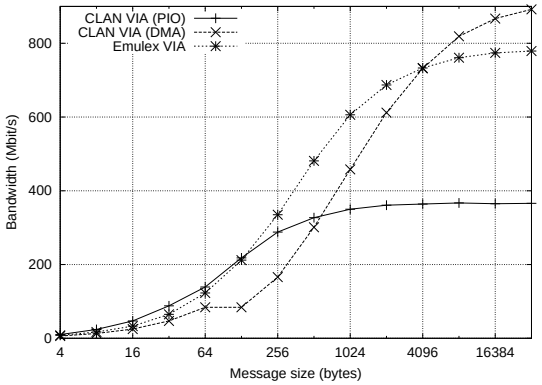

Figure 7. VIA bandwidth vs. message size.

5.4. Analysis

That CLAN VIA has lower latency (and comparable bandwidth) than an ASIC implementation designed specifically for VIA is suprising. Profiling shows that posting send and receive buffers has very low overhead for the Emulex cLAN 1000: about .6 µs. This suggests that performance is limited by high overhead in the NIC. We suspect that the NIC has an embedded processor, which may be limiting performance for small messages.

6. TCP/IP

Although an increasing number of applications are making use of high performance interfaces such as MPI and VIA, the vast majority of distributed applications continue to use TCP sockets.

6.1. Kernel level IP

The simplest way to support IP networking is to use existing support in the operating system. We have written a low-level network device driver for the Linux kernel that works in a similar manner to classical IP over ATM[17]. In our case, IP packets are tunneled over a CLAN connection.

When an IP packet is first sent to a particular host, a CLAN connection is established and used to transmit subsequent packets. Our initial implementation used a distributed message queue (Section 2.1) to transfer the data. When data arrives in the receiving host, a tripwire generates an interrupt, and the interrupt service routine schedules a 'bottom half' which passes the data down into the standard networking subsystem.

The use of the distributed message queue has two disadvantages: (1) the receive buffer has a fixed size and (2) data has to be copied from the receive buffer into the kernel's socket buffers. This was improved upon by using RDMA cookie-based data transfer, as described in Section 2.2. Each host allocates a pool of socket buffers, and sends RDMA cookies for these buffers through a cookie queue to the other host. Data is transferred by DMA directly from socket buffers in the sender to socket buffers

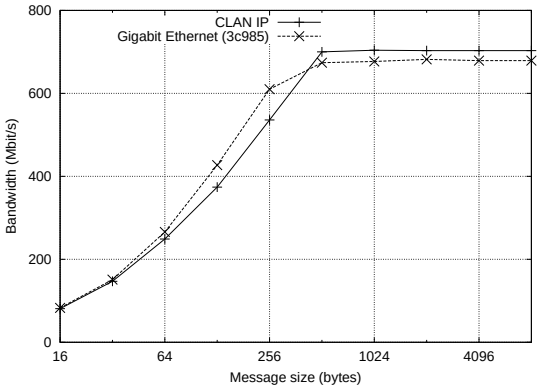

Figure 8. TCP bandwidth for CLAN IP and Gigabit Ethernet.

in the receiver in a similar manner to our VIA implementation. In Figure 6 the send and receive buffers are now kernel socket buffers.

6.1.1 Performance. We have measured the performance of this implementation using the standard TTCP benchmark. Figure 8 shows the results for CLAN IP and a Gigabit Ethernet network using the 3Com 3c985 adapter. The 3c985 is a programmable NIC with two on-board processors. Interrupt coalescing and check-sum offload are used to reduce overhead on the host processor. For both networks, the Linux kernel was configured to allow large receive windows, and 256 KB of socket buffers were used. The 3c985 results were obtained with 9 KB (jumbo) frames, and the CLAN results with an 8 KB MTU.

This configuration exposes the weaknesses of our prototype NIC. Performance is limited by the high overhead of DMA transfers, and we take many more interrupts on the receive side than the 3c985. The 3c985 also benefits from check-sum offload. Despite this, performance for the two networks is very similar up to about 512 byte messages, above which both saturate with CLAN IP slightly faster.

Table 2. Round-trip time for ping over CLAN IP and Gigabit Ethernet.

Network	Test	Ping RTT (µs)	Error (µs)
CLAN IP	normal	78	10
	flood	54	10
3c985	normal	196	37
	flood	216	19

We measured the round-trip time using the standard 'ping' command. The results given in Table 2 are averaged over 100 pings for 'normal' pings (with one second gaps), and over many thousands of pings for the flood ping.

6.2. Accelerating TCP/IP

The performance of the in-kernel TCP/IP support described above is limited by the high overhead of the TCP stack. One solution is to offload some of the protocol onto the NIC, as is done by the 3c985. In the Arsenic[24] project, the NIC demultiplexes incoming data directly into application-level buffers, and the TCP stack is executed at user-level. Overhead is substantially reduced, and a further improvement is gained by using a zero-copy interface. Trapeze/IP[15] also offloads the check-sum calculation and provides a zero-copy socket interface.

Within a local area network, an alternative is to provide a fast path for TCP/IP traffic with a simplified stack that does not duplicate functionality in the network. For example, the CLAN network is reliable and guarantees in-order delivery, so check-sums, sequence numbers, timers and re-transmission are not needed. The use of RDMA cookies for data transfer provides implicit flow control, so management of the TCP receive window could also be removed.

However, this approach is not an option where applications require TCP or the other end of the connection is not in the local network.

6.3. Gigabit Ethernet/CLAN bridge

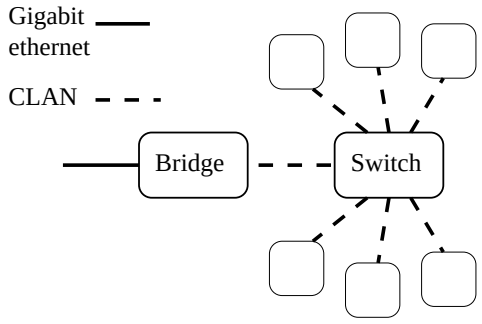

Figure 9. CLAN server room architecture.

Although we can already bridge IP traffic between Ethernet and the CLAN network by configuring a PC appropriately, this solution does not scale well to the high line rates experienced by large server clusters. The Gigabit Ethernet/CLAN bridge connects a CLAN network to the outside world, as shown in Figure 9. Prototype hardware for the bridge has recently been assembled, and is currently in the debug stage. We briefly describe the architecture here.

The main function of the bridge is to demultiplex incoming TCP streams onto CLAN streams which terminate in user-level applications. The IP and TCP/UDP headers of incoming Ethernet frames will be looked up in a CAM to identify the associated CLAN stream. For TCP streams, the sequence number is inspected to determine where the packet data should be delivered in the receive buffer. IP packets that are not associated with a particular CLAN stream will be delivered via a distinguished

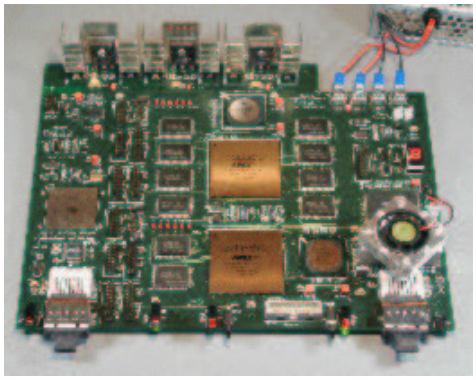

Figure 10. The prototype Gigabit Ethernet bridge.

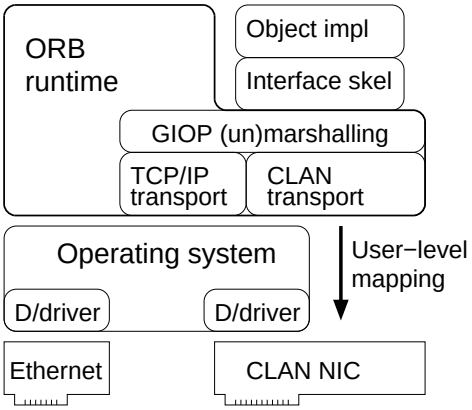

Figure 11. The architecture of omniORB.

stream to the operating system.

The TCP protocol stack is executed at user-level. We have selected the lwIP[14] stack as the starting point for our implementation, which is currently able to exchange packets between CLAN hosts. On the transmit side, complete IP packets are assembled in the application and delivered via a CLAN stream to a staging buffer in the bridge. The bridge will verify key fields in the IP and TCP header, and then emit the Ethernet frames.

7. CORBA

The Common Object Request Broker Architecture[20] is a standard for object-oriented remote procedure call. It simplifies distributed computing by presenting applications with a very high level abstraction of the network. CORBA specifies a language independent object model, a network protocol for invoking requests on objects, and bindings to a variety of programming languages.

The ORB is responsible for providing reliable communication, managing resources such as connections and threads, and providing a number of services. These include management of objects' life cycle, naming, location (including transparent forwarding of requests) and flow control.

7.1. omniORB

omniORB is a CORBA implementation with bindings for the C++ and Python programming languages, developed at AT&T Laboratories-Cambridge. It has been certified compliant with version 2.1 of the specification.

The ORB uses a thread-per-connection model on the server side, which avoids context switches on the call path. The transport interface[18] is flexible and efficient, and transports over TCP/IP, ATM[23], SCI[22] and HTTP (for tunneling through firewalls) have been implemented. The architecture of an omniORB server with the CLAN transport is shown in Figure 11.

7.2. CLAN transport

Data transfer is based on the distributed message queue described in Section 2.1, with tripwires for synchronisation. On the transmit side, the CLAN transport provides a buffer which the marshalling layer marshals a message into. When that buffer fills or the request is completed, it is passed back to the transport layer, where it is transferred into the remote circular buffer either by PIO or DMA. Large chunks of data are passed directly to the CLAN transport, and can be transferred directly to the receiver without first being copied into the marshalling buffer.

On the receive side the unmarshalling layer makes requests to the transport layer for buffers containing received data, specifying a minimum size. The CLAN transport provides direct access to the receive buffer, hence eliminating unnecessary copies. It is possible that the data requested is non-contiguous in the circular receive buffer, so a small amount of space is reserved immediately before and after the buffer, and data copied there as necessary to provide a contiguous chunk to the unmarshalling layer.

7.3. Threads and demultiplexing

omniORB's thread-per-connection model has a number of drawbacks. The principle problem is that a single connection may be serviced repeatedly at the expense of others, until its thread's time slice is exhausted. In addition, a large number of threads are needed if there are many connections, and a thread switch is always needed between requests on different connections. As the cost of the network transport decreases, the impact of these defects becomes more apparent.

Our solution is to adopt a hybrid thread-pool model. A single asynchronous event queue gathers tripwire events from multiple connections, and demultiplexes active connections onto available threads. When more than one connection is active, it is necessary to ensure that sufficient threads are runnable to ensure concurrency is not limited if a thread blocks in the up-call to the object implementation. However, if a thread does not block, it is able to serve many requests before a thread switch occurs.

7.4. Performance

In this section we present some early performance results. The round-trip time for small requests is given in Table 3. The lowest latency reported to date is for Padico[12] on Myrinet-2000 (a 2 Gbit/s technology), which also uses omniORB. We also give results for DCOM (object-oriented RPC for Microsoft platforms) over VIA, taken from [19].

Table 3. Round-trip time for small messages using CORBA and DCOM.

	Hardware and interface	RTT (µs)
CORBA	Fast Ethernet	128
	Gigabit Ethernet (3c985)	180
	CLAN	20
	Padico (Myrinet-2000)	20
DCOM	Emulex cLAN 1000 VIA	≈ 70
	Emulex VIA (polling)	≈ 40

For omniORB we have also measured the maximum request rate when serving multiple clients. The results are given in Table 4. For Fast and Gigabit Ethernet, the ORB was saturated with six clients. The request rate for omniORB over CLAN was greater than 105,000 requests per second with just three clients. Since each request does no useful work, this provides a measure of the total overhead of the ORB and network transport on the server side, which is just 9.5 µs per request.

Table 4. Maximum request rate for omniORB.

Transport	Requests per second
Fast Ethernet	19100
Gigabit Ethernet (3c985)	21600
CLAN	105900

Previous studies have found that ORB overhead is very high[16]. However, the results presented here indicate that ORB overhead can be very low, and considerable improvements are achieved with high performance transports.

8. Future work

Due to the recent closure of AT&T Laboratories-Cambridge, the NIC and Gigabit Ethernet/CLAN bridge are not being developed further. In light of this, the function of the bridge will be emulated on a PC, so that development of CLAN user-level TCP can continue.

MPI performance could be substantially improved by implementing it directly over the raw network, rather than the user-level socket interface. This would allow zero-copy optimisations and reduce overhead.

A number of improvements are being considered for our CORBA implementation, including marshalling messages directly into the receive buffer in the remote application, and using DMA for large messages. The use of an tailored marshalling protocol might also provide an improvement.

9. Conclusions

In this paper we have described the CLAN network, and shown that it's simple, low-level interface supports a wide range of communications paradigms. Each higher-level abstraction is built as a layer of software, without additional support in the network, and without sacrificing performance.

We have found that technologies with more complex network interfaces and protocols are limited to relatively low message rates by the processing requirements on the NIC. We expect the simple hardware model of CLAN to scale more easily to high line rates.

Our MPI implementation has lower latency than comparable interconnects, despite its simplicity. Further improvements can be expected if MPI is implemented directly over the raw network interface rather than user-level sockets.

For both CLAN VIA and in-kernel TCP our implementations give comparable performance to ASIC solutions that have been designed specifically for these protocols. In each case the latency of the CLAN implementation is significantly lower. Further, the CLAN performance would be expected to improve significantly with a proper implementation of the DMA engine.

We have also shown that a fully featured CORBA ORB need not incur the high overhead that has often been associated with it. omniORB over CLAN achieves transaction rates of 105,000 requests per second on our test system, without any modifications to applications.

There has been a trend away from PIO in user-level networks (for example SHRIMP moved to a DMA only model on Myrinet). This is likely to be because it is difficult to manage in the NIC, due to data being pushed rather than pulled. However, we have found that the low latency and overhead of PIO for small messages has been invaluable in the implementation of application level protocols. The combination of PIO for small messages, DMA to offload bulk data transfer and tripwires for synchronisation is a very flexible and efficient model, with a simple scalable hardware implementation.

Acknowledgments

The authors would like to thank members of the Laboratory for Communications Engineering, former members of AT&T Laboratories-Cambridge and particularly Jon Crowcroft for his invaluable advice.

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

References

[1] Emulex cLAN 1000. http://wwwip.emulex.com/ip/products/clan1000.html.

[2] *Intel Virtual Interface (VI) Architecture Developer's Guide*. http://developer.intel.com/design/servers/vi/.

[3] LAM / MPI Parallel Computing. http://www.mpi.nd.edu/lam/.

[4] M-VIA Performance. http://www.nersc.gov/research/FTG/via/faq.html#q14.

[5] M-VIA Project. http://www.nersc.gov/research/FTG/via/.

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

[7] F. Berry, E. Deleganes, and A. M. Merritt. The Virtual Interface Architecture Proof-of-Concept Performance Results. Technical report, Intel Corporation.

[8] 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.

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

[10] P. Buonadonna. An Implementation and Analysis of the Virtual Interface Architecture. Master's thesis, University of California, Berkeley, May 1999.

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

[12] A. Denis, C. Perez, and T. Priol. Towards High Performance CORBA and MPI Middleware for Grid Computing. In *Grid Computing*, 2001.

[13] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects V*, August 1997.

[14] A. Dunkels. Minimal TCP/IP implementation with proxy support. Master's thesis, Swedish Institute of Computer Science, February 2001.

[15] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *USENIX Technical Conference*, 1999.

[16] A. Gokhale and D. Schmidt. Measuring the Performance of Communications Middleware on High-Speed Networks. In *ACM SIGCOMM*, 1996.

[17] M. Laubach. Classical IP and ARP over ATM. RFC1577, January 1994.

[18] S.-L. Lo and S. Pope. The Implementation of a High Performance ORB over Multiple Network Transports. Technical Report 98.4, AT&T Laboratories-Cambridge, 1998.

[19] R. Madukkarumukumana and H. Shah. Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks. In *2nd USENIX Windows NT Symposium*, 1998.

[20] Object Management Group, http://www.omg.org/. *The Common Object Request Broker: Architecture and Specification*.

[21] A. Pant. A High Performance MPI Implementation on the NTSC VIA Cluster. Technical report, NCSA, University of Illinois at Urbana-Champaign, 1999.

[22] S. Pope, S. Hodges, G. Mapp, D. Roberts, and A. Hopper. Enhancing Distributed Systems with Low-Latency Networking. In *Parallel and Distributed Computing and Networks*, December 1998.

[23] S. Pope and S.-L. Lo. The Implementation of a Native ATM Transport for a High Performance ORB. Technical Report 98.5, AT&T Laboratories Cambridge, 1998.

[24] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *IEEE INFOCOM*, April 2001.

[25] L. Prylli, B. Tourancheau, and R. Westrelin. The design for a high performance MPI implementation on the Myrinet network. In *EuroPVM/MPI*, pages 223–230, 1999.

[26] D. Riddoch and S. Pope. A Low Overhead Application/Device-driver Interface for User-level Networking. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001.

[27] 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 Communication. *Journal of Interconnection Networks, JOIN*, 2(3):345–364, September 2001.

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