A Fast Track Architecture for High Performance, Fault-Tolerant Reliable Distributed Group Communication

Brian Whetten
University of California, Berkeley
whetten@cs.berkeley.edu

Todd Montgomery
West Virginia University
tmont@cerc.wvu.edu

Malik Kalfane
University of California, Berkeley
kalfane@cs.berkeley.edu

Keywords: Parallel and Clustered Systems, Transport Protocols, Software Fault-Tolerance

Abstract: This paper proposes a fast track architecture for handling the common case of the UDP/IP protocol stack with very high performance. When combined with one of the new fast, fault-tolerant, reliable multicast protocols (such as RMP or Horus) which are implemented in user space, this architecture will support totally ordered reliable multicast messages with a user-user latency of approximately 200 microseconds, irregardless of the number of destinations of the message. Simpler reliable transport protocols such as TCP/IP will be able to achieve latencies of 10-50 microseconds with this architecture. This architecture will increase the current 13 Mbps RMP has achieved on FDDI rings to a speed limited only by the speed of memory copies on the bus. This latency is one to two orders of magnitude better than that of current totally ordered multicast protocols, yet it is fully compatible with all other IP implementations.

The main points of the architecture are an early demultiplexer called by the hardware device driver and a fast track for the common case packets which almost completely bypasses the operating system. This performance is achieved without sacrificing protection between user processes.

1 Introduction

Totally and causally ordered, reliable broadcast and multicast protocols have existed for quite some time [ChMa84], and provide a powerful tool for programming distributed systems and distributed databases [Chang84]. New applications such as Computer Supported Cooperative Work (CSCW) programs, groupware systems and shared tools can also benefit greatly from this service. In the past, cluster computing applications have not been supported well by these protocols, because they have had problems with performance, efficiency, and/or scalability. It has become a widespread belief that these are inherent problems with a totally ordered reliable multicast protocol [RaLi93]. In part, this concept resulted from the fact that in the past multicasts had to be implemented as a series of unicasts to each destination. Recent developments such as the IP Multicasting standard [Deering89] now allow a multicast datagram to be sent to multiple destinations over an internetwork. In the case where all destinations are on the same LAN, one multicast packet to all of them costs the same as a unicast packet to just one.

The Reliable Multicast Protocol (RMP) was the first protocol to demonstrate that a fault tolerant, fully distributed, totally and causally ordered, atomic multicast protocol could deliver messages with better latency and throughput than even TCP/IP, for groups that consist of two or more destinations [WhMoKa95]. For example, on 8 SparcStation5's on a 10 Mb/sec (1250 KB/sec) Ethernet, the current implementation's throughput to a single destination is 1070 KB/sec, or 86% of the network capacity. In this case, latency is 4.0 milliseconds (ms). In this environment, unlike all other unicast systems and unlike almost all other totally ordered reliable multicast systems [ADKM91, AMSM92], both the throughput and latency of packets stays roughly constant as the number of destinations increases. For 8 destinations, the RMP aggregate throughput is 7.4 MB/sec (5.9 times the capacity of the Ethernet) and the minimum length packet latency is 4.9 ms.

The total ordering and fault tolerance that protocols such as RMP provide would be of great benefit to cluster computing and network of workstations (NOW) applications, except that the latency of packets is still too high. Because of this, groups such as the NOW project are implementing completely new protocols such as Active Messages, in order to provide low latency communication. However, in examining the performance of RMP and other transport protocols, including very standard (and supposedly highly optimized) protocols such as TCP/IP, it is evident that in LAN environments, the way that the IP stack is implemented in the kernel is the limiting factor of performance, rather than any inherent problem with any protocol specifications. For example, RMP is implemented as a library in user space, running on top of UDP/IP Multicast. Even though RMP is a relatively complex protocol, only 4% of the processing time for minimum length packets is spent in RMP, with the rest being spent in system calls.

Kernel implementations have typically been designed for generality, interoperability, and clean separation of layers, rather than for performance. While this was appropriate in the past because of the unreliable internetworks TCP/IP and UDP/IP operated on, the common case for modern networks (particularly the LAN environments that NOW projects run in) is of packets that arrive reliably, in order, and all in one segment [Dalt93]. In addition, most implementations have poor performance because the receiver doesn't demultiplex packets early enough, because packets are copied multiple times at the sender and the receiver, because multiple context switches occur for each packet, because the packet checksum is computed separately, and because buffers are handled inefficiently.

The biggest bottleneck in the throughput of transport protocols today is usually the multiple times a packet is copied. Three very good solutions have been proposed, all of which achieve a single copy per packet [Brust94, Dalt93, MBH94]. Given a single copy protocol, we describe a fast track architecture that will, in many environments, decrease the latency of the TCP/IP, UDP/IP, and UPD/IP Multicasting protocols from the current 2-5 ms to 10-50 microseconds (μs). For protocols such as RMP that are built in user space on top of UDP, the latency will be reduced to 100-250 μs. At the receiver, this fast track demultiplexes common case packets at the time of the device driver interrupt, processes the headers in an efficient manner, calculates the checksum while the single copy is being performed, uses shared buffers to avoid multiple copies, and avoids all timer calls such as gettimeofday(). At the sender, the fast track sends common case packets directly to the network interface, filling in the headers with mostly cached values.

The primary novel advantages of our architecture are that it is applicable to a wide range of network interfaces, it will work with any of the proposed single copy approaches, it can provide full inter-process protection with only a moderate latency penalty, and it shows how TCP/IP, UDP/IP and protocols based on UDP/IP can provide very low latency. This is of particular value because our architecture is compatible with other implementations, because TCP/IP and UDP/IP are used so widely, and because it will allow cluster computing applications to use totally ordered, fault-tolerant reliable multicast systems with a latency of 100-250 μs.

Section 2 describes the previous work that has been done in this area. Section 3 describes options for achieving single copies of the data in packets, which is critical for high throughput. Section 4 describes the fast track architecture at the receiver. Section 5 details the architecture for the sender and discusses some protection issues of the fast track. Section 6 lays out the calculated performance of our system. Finally, section 7 presents our conclusions and plans for future work.

NOTE TO REVIEWERS: While this paper is based on analysis, we are currently implementing this architecture in HP-UX. We should have a fully working prototype by the time of publication, and will replace section 6 with our measurements of this prototype.

2. Previous Work

Craig Partridge [PaPi94] has shown some ways to speed up typical implementations of UDP/IP, including folding the checksum into the copy, and implementing one-behind caches for incoming packets. While very little other work has been done on speeding up UDP/IP, a good deal has been done with TCP/IP which would apply to UDP/IP as well. Van Jacobsen [Jacob93] points out that in traditional TCP/IP implementations, interrupts and memory copies are very costly and are the main reason why TCP/IP has not been a very high performing transfer protocol. Old implementations of TCP/IP used two interrupts per packet sent or received, and much more memory traffic than needed. He proposes a new implementation which needs 37 instructions and 7 memory references for IP to forward a packet, and under 60 instructions and 22 memory references to process the protocol in a TCP datagram. He also shows how the checksum can be computed while copying the data, for very little (if any) cost. He explains that the bottleneck is at the receiver, since it has to demultiplex incoming packets and transfer data, whereas the sender only has to transfer data. Finally, he proposes to make advantage of the fact that, in high speed data transfers, consecutive packets almost always demultiplex to the same place. Many of these ideas are incorporated in our architecture.

Brustoloni [Brust94] used exposed buffering, which consists of buffers shared between applications and the network interface, to implement a low latency ATM network interface. This was done on DECstation 5000 workstations running the Mach 3.0 operating system, which made it easier to use techniques such as exposed buffering to improve the network interface performance. He achieves a one way latency of 85 microseconds for a 36 byte message.

Dalton et al. [Dalt93] implemented a network interface card, called Afterburner, as a test of the WITLESS architecture proposed by Van Jacobson, which is a network interface that supports single copy protocols. This card is designed for the HP 700 series of workstations. It occupies a slot in the workstation's graphic bus, and contains 1MB of VRAM which is mapped into the processor's memory. They have implemented a single copy version of TCP on Afterburner which achieves a throughput of 25 MB/s (210Mb/s) on 14KB packets, which is close to the interface to memory copy rate of 32MB/s.

Minnich et al. propose a zero-copy ATM interface called MINI (Memory Integrated Memory Interface). They aim at a bandwidth of 1.2 Gb/s and a latency of 1.3 microseconds. Applications can have direct

access to pages of memory used for network I/O, which allows bypassing the operating system on both sending and receiving packets. Standard networking software such as TCP/IP can still go through the operating system. The big gains in performance are obtained by reducing the number of data copies needed, bypassing the operating system, and integrating the network architecture into the memory hierarchy, rather that the I/O hierarchy. They are aiming for a single-copy implementation of TCP that would achieve 40MB/s throughput. The main problems which have not been addressed are those of cache coherency.

Richard Martin [Mart94] implemented an Active Message layer, which is a thin, highly optimized communication layer that aims to deliver the minimum latency and the peak bandwidth of the network hardware to user programs. His implementation, called HPAM, is an Active Message layer for a network of HP workstations. Using HP9000/735 workstations equipped with Medusa FDDI interface cards, HPAM achieves a roundtrip time for a 20 byte payload of 29 microseconds, and a maximum bandwidth of 12MB/s. In addition to supporting point to point communication, HPAM also supports group communication primitives for things like barrier synchronization. HPAM gets very low latency while maintaining reasonable protection by associating a separate network interface state with each process and allowing the currently swapped in process to have direct access to the interface.

Kay and Pasquale [KaPa93] made fine grained measurements of TCP/IP in Ultrix 4.2a, and showed that a large portion of the time processing packets in this implementation is spent in context switches and other overheads that do not change much as the length of the packet changes. This helps support the need for a fast path through the OS.

3 Single Copy Options

As memory copies are expensive and are the main limiting factor to network communication throughput, the fast track architecture needs to limit to only one the number of data copies on both the sender and the receiver. Depending on the network interface, there are several options for accomplishing this. The main problem that needs to be solved to achieve a single copy system is that the data must be copied from the network interface directly into user space (and vice-versa on the sender side), rather than traditional implementations that first copy the data to OS space, and then have the OS copy the data to user space.

This single copy can be done either through programmed I/O, in which the CPU performs the transfer from memory to the I/O adapter (and vice-versa), or through DMA, in which the network card directly writes/reads data to/from memory, bypassing the processor. With DMA, one has to worry about cache

coherency with respect to DMA transfers, and the overhead of setting up the DMA transfer. As pointed out by Dalton et al. [Dalt93], depending on the architecture, DMA may not be the most efficient mechanism. In their implementation on HP 9000/700 machines, programmed I/O proved to be more faster.

One way to solve the single copy problem is to have operating system level support for exposed buffering, as was implemented by Brustoloni [Brust94, DrPe93]. In this method, exposed buffers are shared between the sending or receiving application and the network interface, so that no data movement is required. Each application shares a set of buffers with the kernel. The trade-off in using exposed buffers is that applications need to worry about their buffer usage, and need to use special calls to allocate memory. While this solution will work with most interface cards, it does require extra modifications to the OS. A criticial issue with shared buffer transfers for DMA cards is how to determine which user's buffer the interface should copy an incoming packet into. We solve this by noting that low latency is only critical for the currently swapped in process. Given this, we have the interface write to a shared buffer owned by the currently active process. If this guess is wrong, the fast track then moves it over to the correct destination.

Dalton et al. [Dalt93] use another approach to get a single copy architecture. They include 1MB of VRAM on the network interface card, in which incoming and outgoing packets are stored. Programmed I/O is then used to read/write data directly from/to user space, without having to go through kernel space. The advantage of this method is that no special operating system support is needed, but the trade-off is that the network card needs to provide its own memory. In this case, you do not need to use shared buffers for the fast track architecture.

Yet another approach to get a single copy architecture is proposed by Minnich et al. [MBH94]. Their solution, called MINI (Memory Integrated Network Interface), is an ATM interface that resides on the local memory or graphics bus, rather than on the I/O bus. This allows the network interface to use DMA to efficiently read/write data from/to user space. On the receiving side, virtual channel identifiers (VCI) are used to determine where the incoming packets should go, and they are automatically copied there. This allows for very high throughput and very low latency network communication, with claimed performance expectations of 1.2 Gb/s throughput, and 1.3 microseconds latency. As with Afterburner, the fast track does not need to use shared buffers with this type of network interface. Furthermore, MINI makes much of the demultiplexing code redundant due to it's use of VCI's to automatically select which user's space the incoming packet should be copied to.

Our architecture assumes that one of these methods will be implemented in the host system. If not, throughput will drop by approximately a factor of two for large packets. Latency of small packets will not be affected as much, as the cost of copying isn't as significant in this case.

4 Receiver Fast Track Description

Besides copying the data multiple times, the primary problem with current IP and IP Multicasting implementations are that they are optimized for the general case instead of for the common case. Because of this, they waste a lot of time processing the packet headers, they calculate the checksum in the OS, they do slow buffer management, they use expensive system calls like gettimeofday(), and they incur multiple context switches. To get around these problems, we propose a "fast track" for the common case packets. At the receiver, the fast track handles a packet only if it meets the following conditions:

- There is a socket established for this packet (UDP or TCP)
- It is destined for this host (IP)
- It is either a UDP or TCP packet (IP)
- The correct version of protocol is specified (all)
- It is not a control packet (TCP)
- It arrives without an error (all)
- It arrives in order (TCP)
- It has no variable length options (IP, TCP)
- The packet is not fragmented (IP)

In the above list, the protocols which have to make the given checks are specified in parenthesis. When packets arrive for an application that do not meet the common case, the fast track is disabled until after these packets have been handled. Packets that arrive while the fast track is disabled may meet the above conditions yet not be handled by the fast track. At the receiver, our fast track architecture requires changes in the organization of the OS, an early demultiplexer at the receiver, fast processing of the common case packet headers, and a set of smaller optimizations like folding the checksum into the memory copy.

4.1 Receiver Fast Track Organization

The receivers in current implementations of IP are very inefficient, typically requiring a packet to be involved in a hardware interrupt, a software interrupt, multiple context switches, three queues and two IPCs before being delivered. Receivers are generally accepted as the bottleneck in communication latency

[Jacob93, Mart94, Traw93]. Figure 4.1 shows a typical implementation of an IP receiver. This design is very clean and well layered. For connections which have a lot of packets which do not meet the common case, this will be relatively efficient. However, in many environments such as most LANs, virtually all packets will meet the common case described above [Dalt93, Jacob93, Brust94]. For example, a FDDI or ATM LAN should only corrupt one in 10^9 packets. ATM networks guarantee that packets will not be duplicated or delivered out of order. While the recent decision to adopt rate based flow control rather than credit based flow control will make congestion more difficult to control for Available Bit Rate (ABR) traffic, as the width of the communication pipes increase, this becomes less and less of a problem.

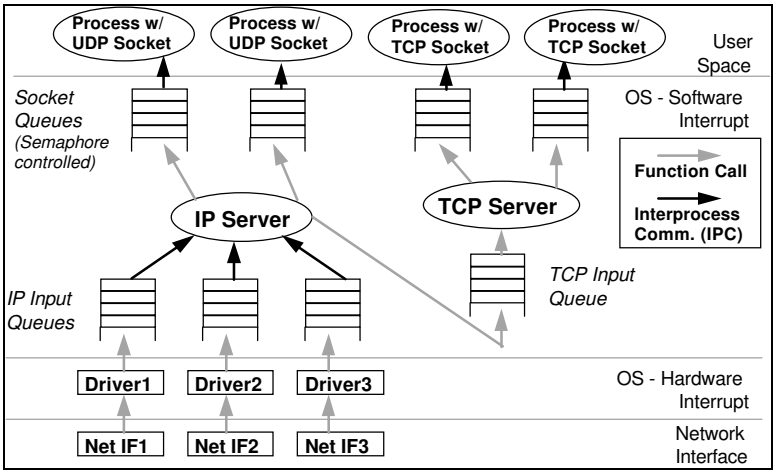


Figure 4.1: Current Receiver Organization

The fast track for common case packets completely avoids context switching into the OS. Figure 4.2 shows the proposed architecture for received packets. In this design, the packet demultiplexing and fast track processing is done in a routine called directly by the device driver. This is feasible because the overhead of a hardware interrupt on a RISC machine is typically around 1000 instructions, while the fast track code is between 200 and 300 instructions. If packets meet the common case, they are sent directly to the user space. Packets that do not meet the common case are sent to the normal OS handling routines, either by the demultiplexer or by the fast track routines. For low latency, avoiding the OS in this way is absolutely critical, as a software interrupt typically costs a minimum of 30 microseconds, and often much more.

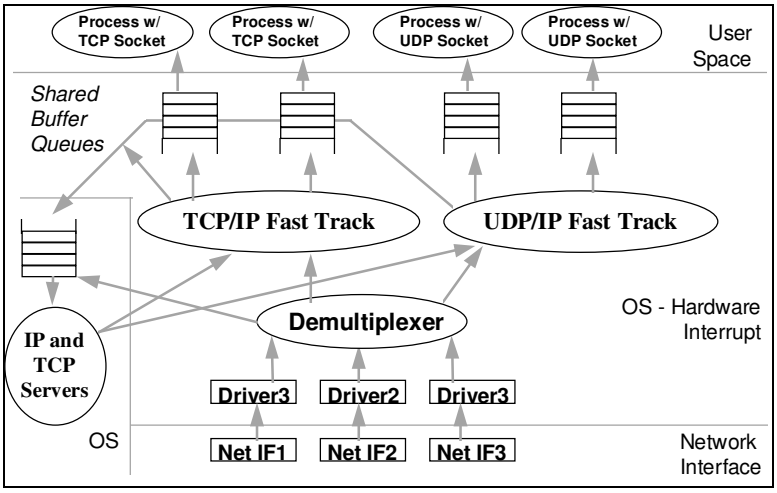


Figure 4.2: Modified TCP/IP Receiver Path

4.2 Early Demultiplexer

The most important element of the receiver fast track is the early demultiplexer. This is a general piece of software that allows hardware device drivers to determine the destination of a packet and send it there immediately. A packet can be sent to a queue in a user process or in the operating system. The demultiplexer can handle a variety of transport protocols without the need for recompilation of its code, and it supports both IP and IP Multicast packets. The data structures for the demultiplexer consist of a table with one entry for each socket and two hint pointers to the most recently used entries in the table. Each socket uses the OS to register a set of identifying fields with the demultiplexer that allow the demultiplexer to determine the destination of a packet without protocol specific code. The socket also registers pointers to two queues for the user, and a flag as to whether or not the socket is a multicast one. These fields are listed in figure 4.3.

The way that most sockets identify themselves is through their {Source IP Address, Destination IP Address, Source Port, Destination Port, Protocol Type} tuple. These fields will be included in incoming datagrams. The other way that a socket can identify itself is through a virtual circuit identifier (VCI). This is only applicable if the network interface is using virtual circuits. In this case, the socket registers the VCI and the VCI offset into the header. VCIs are a very efficient way of demultiplexing packets, but are only applicable to ATM networks.

Entry Name	Bytes	Description
Dest IP Address	4	Part of socket identifier for datagrams
Source IP Address	4	Part of socket identifier
Dest Port	2	Part of socket identifier
Source Port	2	Part of socket identifier
Protocol Type	1	IP protocol type field
Dest Port Offset	1	Offset of destination port into header
Source Port Offset	1	Offset for source port
Multicast?	1	Is this a multicast socket?
User Queue	4	Pointer to destination queue
Free Queue	4	Pointer to free queue
VCI Offset	1	Offset for VCI (255=No VCI)
VCI	2	Virtual circuit identifier

Figure 4.3: Demultiplexer Table Entries

For non-ATM networks, packets must be demultiplexed off of their IP, UDP, and TCP headers. The IP Address and Type fields are in the IP header, and the Port fields are in the TCP or UDP

header. Because of this, the socket may also register an offset into the header for each of these ports. This allows multiple transport level protocols to be supported without recompiling the demultiplexer routine. A complication of these port fields is that they will only be in the first segment of a fragmented IP packet. This is solved since the common case the fast track deals with is for non-fragmented packets.

In addition to the identifying fields, each socket has a multicast flag associated with it. Most sockets will not be multicast capable, and the check of this flag won't impose much overhead. Multicast sockets require that all of the entries in the demultiplexer table be checked, since a packet may need to be delivered to multiple destinations. Otherwise, the first matching entry in the table is sufficient.

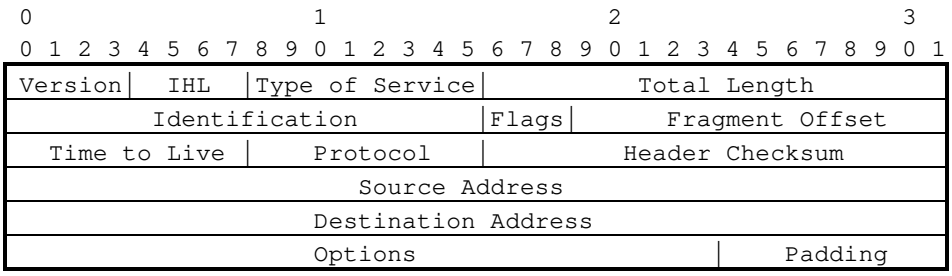
The final piece of information stored in each entry of the demultiplexer table is the location of two queues for that socket. One is the destination queue, where incoming packets will be placed. The second queue is a free buffer queue, where free buffers for the user are stored. Most NI's will use these buffers as the destination for the single copy they perform. In a NI like Afterburner or Medusa, this copy will occur directly into the user space after the packet is finished being handled by the user space library. A NI like MINI [MBH94] can pass the receive buffers from the free pool all the way down to the network interface, where they are written directly to when the packet is received.

```
If packet is fragmented
    | Pass packet to OS
Get type field from header
Based on type field, get offsets
If VCI offset is valid
    | Get VCI field from header
    | Check two hints for VCI match
    | If no match
        | Search table for VCI identifier
    | If match not found
        | Send to OS (slow path)
    | Else
        | Put in user's input queue
Else
    | If packet is fragmented
        | Send to OS (slow path)
    | Get IP address and port fields from header
    | Check two hints for identifying tuple match
    | If no match
        | Search table for identifying tuple match
    | If match not found
        | Send to OS (slow path)
    | Else
        | Put in user's input queue
```

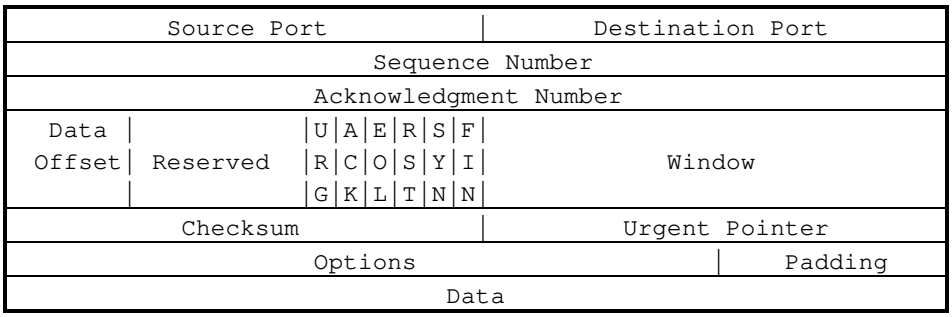
Figure 4.4 Demultiplexer Algorithm

For a protocol such as TCP/IP, demultiplexing packets early is the biggest performance improvement besides avoiding multiple copies. While management of the demultiplexer entries is handled by the OS, when a packet arrives the demultiplexer allows it to skip right over the OS into the user space. The demultiplexer is accessed directly by the hardware interrupt for each driver. When a packet arrives at the driver, it calls the demultiplexer routine. The demultiplexer performs the algorithm in figure 4.4 This routine uses the type field in the IP header to determine which transport protocol the packet is part of, and uses this to find either the port number or the VCI in the transport packet header. Using either of the two identifier types, it then checks the two hints for the most recently used table entries. If these both

IP HEADER



TCP HEADER



UDP HEADER

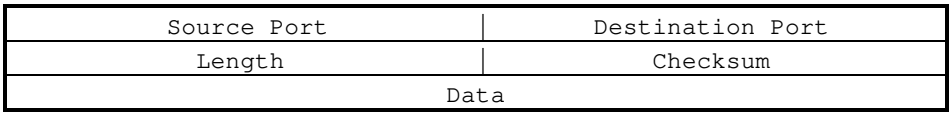


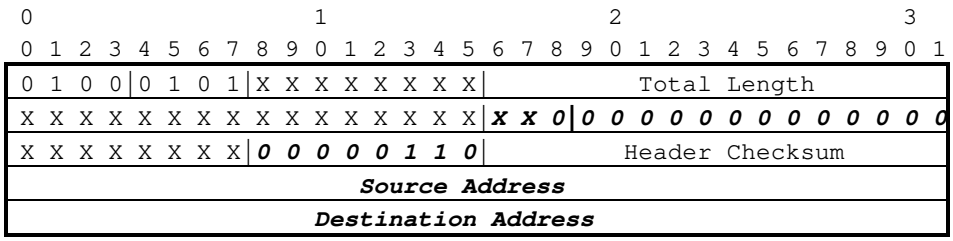
Figure 4.5 TCP, UDP and IP Packet Headers

fail, a full search of the demultiplexer table is done. If this also fails, or if the packet is fragmented, the packet is sent on the slow track to the operating system, where it is handled by the standard routines.

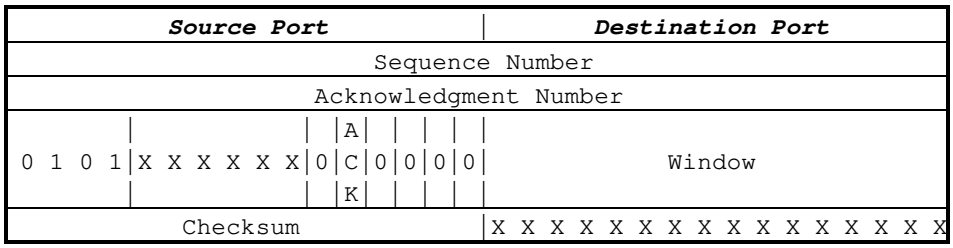
4.3 Common Case Packet Handling

For the common case of packets that the fast track is handling, packet processing can be made very efficient. Van Jacobson [Jacob93] has claimed that by doing TCP and IP header processing at the same time, the processing for a common case packet can be done in less than 100 instructions at the both the sender and receiver. Since UDP is much simpler than TCP, this also applies to UDP packets. We sought to verify that this claim would hold in our framework. To this end, we implemented a pair of packet processing routines that just handle the common case, one at the sender and one at the receiver. The results are given in the performance section. The key observation here is that for the common case, most of the header fields for a given TCP/IP packet stay constant or change in simple ways. Therefore, most of the fields at the sender can be filled in with cached values, and the receiver just needs to verify the incoming fields against the same cached values. Figure 4.5 shows the IP, UDP and TCP packet headers. Figure 4.6 shows which of these fields the receiver has to verify, which fields are handled by the demultiplexer, and which fields the receiver actually has to deal with. For the common case, a TCP reciever only has to deal with 7 fields, and a UDP

IP HEADER



TCP HEADER



UDP HEADER

X = Don't care
***Text* =** Fields handled by the demultiplexer
`Text = Fields` handled the fast track receiver routine

Figure 4.6: Common Case Receiver TCP, UDP, and IP Headers

receiver only has to deal with 4. Section 6 describes each of the steps in the sender and receiver packet processing in more detail, along with an analysis of how many instructions the code takes.

4.4 Other Fast Track Issues

There are a set of smaller issues that must also be dealt with in order to get high performance in the fast track architecture. These are packet checksums, timers (for retransmission and round trip time estimation), changes in the TCP user interface, and allowing for blocking select calls at the receiver. Calculating a checksum over a packet is normally an expensive operation. However, Van Jacobson has shown that this can be folded in with a memory copy on most modern architectures without reducing the copy rate [Jacob93]. In addition, many network interfaces now include support for checksum calculation. In our performance calculations, we assume that one of these two approaches has been taken.

A second issue that must be dealt with is timers. A system call to gettimeofday() costs 18 microseconds on a HP 735 series [Mart94], which makes this prohibitive. Some architectures such as the HP PA-RISC now include an interval timer which is stored in a register and updated every cycle [Mart94]. If this option

is not available, then it is not feasible to timestamp each packet that is sent. Instead, we propose that the TCP server in the OS wake up every 10-50 ms, mark all of the fast track packets that have been sent in that interval, and resend any packets that have been waiting for acknowledgment for more than one interval. We call the period between server polls an *epoch*. This approach to timeouts is feasible because while it will make timer estimations inaccurate, current TCP implementations rarely see a timer for a dropped packet go off, particularly under the conditions that produce the common case we are optimizing for [Paxs94]. Most dropped packets are detected not by a timer, but by the receiver. When this happens, the reciever sends back an ACK which tells the sender it needs to resend a packet.

When it wakes up, this TCP timer routine can also do the round trip time estimation on common case packets. For each packet that has been sent that spans two or fewer epochs, one half of one epoch length is averaged in to the estimation. For packets that span three or more epochs, the estimation for that packet round trip time is (# epochs-1.5)*epoch_length.

Our architecture supports two user interfaces: the traditional BSD socket interface, and a slightly modified version of the interface which supports shared buffers. Shared buffers allow even Ethernet cards to achieve packet transmission and reception with only a single copy. However, this changes the user semantics slightly. When a packet is sent to the network, the user no longer maintains ownership of the buffer, although this is not enforced. Also, the user must request these buffers from the OS with a special system call instead of allocating them with malloc(). Finally, the receive call for a socket does not copy the data to a buffer, but returns the pointer to a buffer that has the data in it. These shared buffers are needed to obtain maximum performance on traditional network interfaces, although newer interfaces such as Afterburner [Dalt93], MINI [MBH94], and Medusa [Mart94] do not require this.

The last change for low latency performance is that the user must poll the OS instead of using blocking select calls. Blocking select calls require intervention by the operating system and pay a very high overhead. If allowed, these polls to the incoming queue can also prompt a poll to the network interface. This avoids the cost of having a hardware interrupt for each incoming packet. However, this violates some of the protection that UNIX offers, and so may not be feasible.

5 Sender Fast Track Description

While the receiver is the primary bottleneck in current communications, the sender routines also require a fast track, or they will become the dominant bottleneck. At the sender, the fast track handles a packet if it meets the following common case conditions:

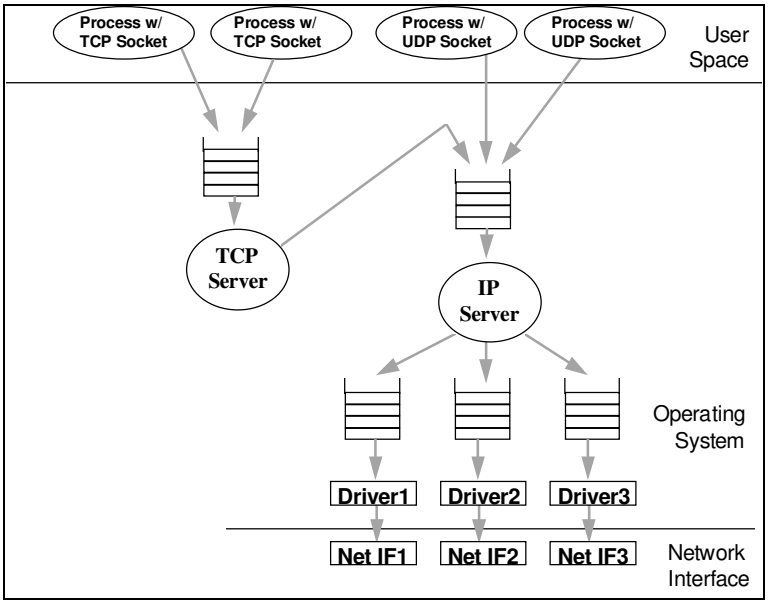


Figure 5.1: Typical Sender Organization

- There are no packets already waiting to be sent (all)
- The packet does not need to be fragmented by the IP layer (IP)
- There is already an outgoing connection for this socket (UDP, TCP)
- This connection has already resolved the link address using ARP (IP)

Like the receiver, if any packets are sent that do not meet the common case, the sender fast track is disabled until after these packets have been handled. Changes at the sender include changing the organization of the send path, handling the processing for the common case packets, caching of ARP entries, and changing the OS to minimize the crossing of protection boundries.

5.1 Sender Fast Track Organization

Like the receiver, the typical sender path is optimized for the general case instead of the common case. Figure 5.1 shows a typical organization of the sender path. For TCP packets, this involves a context switch into the operating system, three queues, and three handler routines. Figure 5.2 shows the fast track sender organization. When an application calls a socket routine to send a TCP or UDP packet, a fast track handler is first called. This handler checks to see if the common case conditions are met for this packet. If so, it fills in the necessary header fields from a cached copy of these fields, handles the minimal processing needed for these packets, and sends them out.

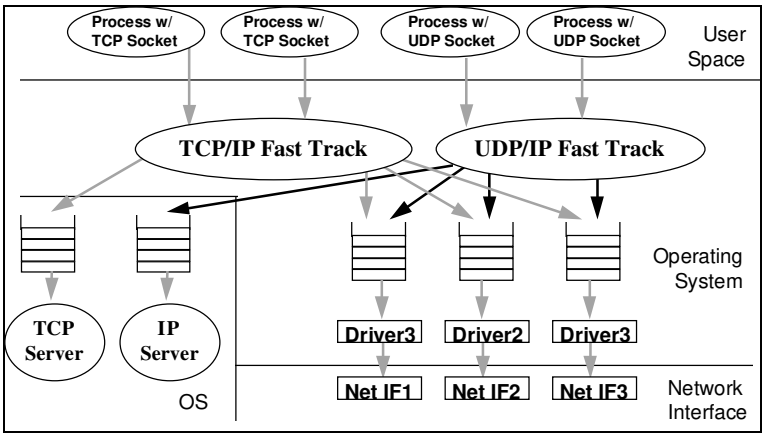
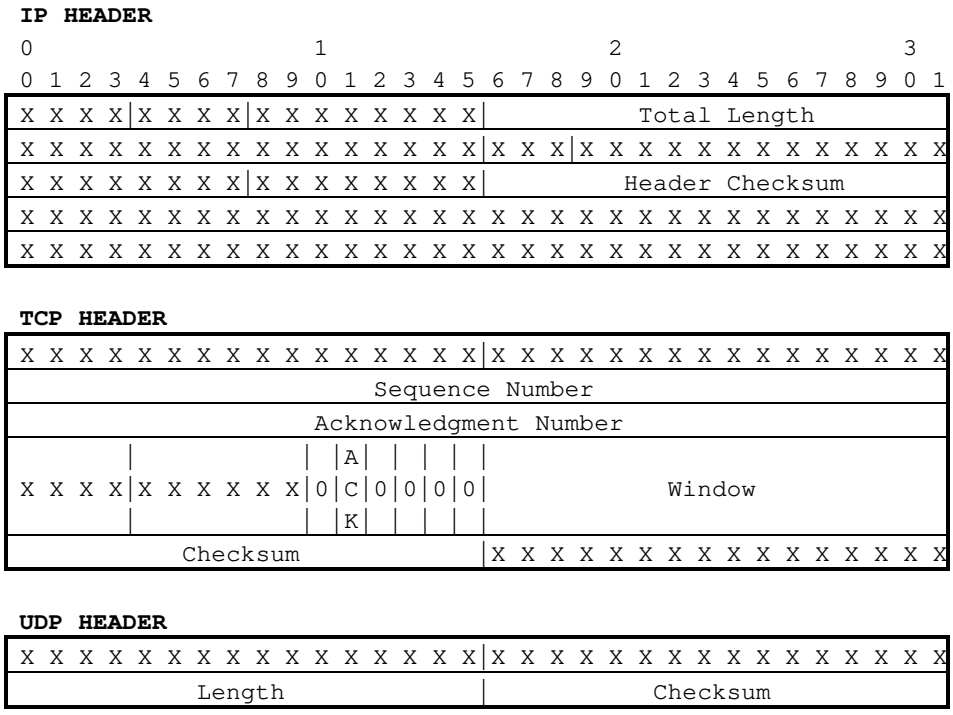


Figure 5.2: Fast Track Sender Organization

5.2 Common Case Processing



X = Cached from last packet sent

Text = Fields handled by the fast track sender routine

Figure 5.3: Common Case Sender TCP, UDP, and IP Headers

Like the common case header processing for the receivers, the sender has to do very little work. Almost all of the fields in the new headers are identical to the last packet sent over this channel. Figure 5.3 shows the common case headers for IP, UDP, and TCP. IP has two fields that must be filled in, TCP has five, and UDP has two. In addition to these fields, the link level header must also be filled in. To do this, a pointer to the ARP cache entry for this destination is added to the information for this socket. This information is quickly looked up at send time and filled in as well. By handling all of the headers at once, a single buffer of known size can be allocated to handle all of the headers.

5.3 Protection

The fast track architecture achieves high performance without sacrificing protection between users. The two potential protection problems with the fast track architecture are in the shared buffers and in allowing the sender to avoid context switching into the operating system. Shared buffers do not compromise inter-user protection. The buffers are only shared between a single user and the operating system. At the reciever,

because the demultiplexer operates as part of the device driver, it is protected code. Since only one user can access each buffer, and because the code to fill in the buffer is secure, the privacy of the buffers is maintained. A user can violate protection of his own buffers, by modifying the contents of them after sending, but can not modify the buffers of anyone else.

For high performance, a big win could be gained by avoiding the context switch at the sender. This could be avoided if a protected user library had direct access to the hardware driver. In addition, the receiver would like to be able to directly poll the hardware driver under times of heavy load, to avoid the costs of fielding a hardware interrupt, which has an overhead of around 10 microseconds on a HP 735 [Mart2/94]. Both of these pieces of the user library must be protected from corruption by the user in order to provide full protection. As alternatives to putting the code in the OS, this code can be left unprotected, or an alternative mechanism can be used to protect the code. Mechanisms exist in some architectures such as the PA-RISC hardware that allow for protected code and data without having to execute a kernel trap and the associated context switch [Mart94]. If efficient, these mechanisms would solve this problem by allowing the user library to share the same protections as the operating system while not requiring a full context switch. If not, the OS will have to allow some protection holes, or pay the latency penalty of context switching.

6 Code Analysis

We wrote code for the TCP/IP send and receive functions, as well as for the demultiplexer. We compiled the code on a Sun Sparcstation, using gcc, with the '-O4' option (which is the highest optimization option), and the '-funroll-all-loops' option (which causes all the loops to be unrolled). For the common case, the send function needs approximately 90 instructions, the demultiplexer 45 instructions, and the receive function 130 instructions, as described below. This does not include the cost of copying data, which we consider separately. The breakdown on the number of instructions used in the send and receive header processing is shown in figures 6.1 and 6.2.

On the sender side, the following actions occur. First, the sender makes sure that is will not have to block the packet due to its send window not being large enough. Next, it creates a copy of the header from the cached template for this socket. It then has to fill in seven of the fields. Some of these, such as the acknowledgment number and sequence number, are straightforward. The window field is more complicated, because it requires implementing the silly window avoidance algorithm. Finally, the header checksum is computed and filled in, and the packet is shipped off to the network interface, possibly via the OS. Assuming 50 MIPS CPU performance, this will take 90 instructions / 50 MIPS = 1.8 microseconds.

At the receiver, the following actions occur. First, the library checks four of the fields of the header to verify that this is a common case packet. If not, it is sent to the OS for slow track handling. Next, the receiver computes the length of the data field in the packet, and updates the next expected sequence number. The IP header checksum is then computed and verified. The TCP checksum will have been computed by this time, either by the NI or in the copy. This is verified against the TCP checksum field. If either of these fail, it is reported to the OS. At this time, the receiver knows that this is a valid common case packet. It uses the TCP acknowledgment and sequence number fields to update some of its state variables and its congestion control window. It then checks to see if it needs to send an ACK, which is necessary if more than a threshold number of unacknowledged bytes have been received. The data field of the buffer is then passed to the user queue, and the receiver checks the send queue to see which packets have been acknowledged. These packets are freed. In total this takes 130 instructions. Assuming 50 MIPS CPU performance, this will take 130 instructions / 50 MIPS= 2.6 microseconds.

Instruction Count	Description
19	Verify that this is a common case packet
6	Calculate the length of the data
24	Calculate and check the header checksum
10	Update state variables (window size, sequence number, etc.)
8	Update congestion window for Van Jacobson congestion control
35	Free up packets which have been acknowledged
18	Check to see if we need to send an acknowledgment
12	Send the packet to the user

Figure 6.1: Packet Handling at the Receiver

Instruction Count	Description
9	Verify that we can immediately send this packet without fragmenting it
8	Verify that we already have a link address for this destination
26	Fill in the cached values for most of the header fields
12	Fill in length, sequence number, and acknowledgment number
10	Calculate window we should advertise, update state variables
24	Calculate and fill in the header checksum

Figure 6.2: Packet Handling at the Sender

In the common case for a high speed data transfer, as described by Jacobson [Jacob93], we can assume that consecutive packets demultiplex to the same location. With that assumption, the demultiplexer takes 45 instructions to execute. In the worst case it will take at least 105 instructions or longer, proportional to the size of the table. In the common (best) case, this routine then takes 45 instructions / 50 MIPS = 0.9 microseconds.

6.1 Calculation of Total Latency

According to [Drus93], on a Sparcstation-10, we can expect sustained throughput of 44 MB/s on read, 42 MB/s on write, and 27 MB/s on copy. We use these figures to estimate the latency of our system.

On the sender side, the latency will consist of:

- Computing and filling in the header: We need 1.8 microseconds to compute the header fields, and assuming a 42MB/s write rate, we need 40/42=1.0 microsecond to write the header.
- Data copy: Assuming a 27 MB/s copy rate, we need 24/27=0.9 microseconds to copy the 24 byte data payload per packet.

Hence, the latency to send a packet is 3.7 microseconds.

On the receiver side, we get the following latency calculation:

- Demultiplexer code: From above, this will take 0.9 microsecond to execute.
- Header cache miss: We need to load the packet header in cache in order to process it. Assuming a 44MB/s read rate on 40 bytes, we need 40/44=0.9 microseconds to load the header.
- Data copy: We need to copy the packet (64 bytes). Assuming a 27 MB/s copy rate, we need 64/27=2.4 microseconds for the data copy.
- Header processing: Once in user space, we need 2.6 microseconds to process the header, as described above.

Hence, on the receiver side, we need 6.8 microseconds to process the packet.

The total latency of our system is 10.4 microseconds, not including the time spent in the network interface or on the network itself. In addition, there may be time spent in the hardware interrupt at the receiver and in the context switch at the sender. While we have discussed possible solutions to these problems, they will not be practical in many systems, and will probably add an additional 30-40 microseconds. Therefore, we expect to see between 10 and 50 microsecond latencies for UDP/IP, TCP/IP, and UDP/IP Multicasting packets with our fast track architecture.

6.2 Latency for RMP

Given a UDP/IP service with a latency of 10-50 microseconds, we expect RMP to be able to provide fault-tolerant, totally ordered messages with a latency of approximately 200 microseconds. Profiles of RMP over a lightly loaded Ethernet on SparcStation 10's have shown that RMP spends 160 microseconds processing both a message and its acknowledgement [Mont94]. Of this, 40 microseconds is spent

processing the data packet itself, and the other 120 microseconds is spent on the acknowledgement. In sending and acknowledging a totally ordered packet, RMP has to put two UDP/IP packets on the wire. From the calculations in the last section, this is 20-100 microseconds. When added to the 160 microseconds RMP spends processing the two packets, this is a total of 180-260 microseconds. In a group of two senders, this would increase the throughput of RMP in terms of messages per second from 250 to over 3750.

7 Conclusions

In this paper, we have described a fast track architecture that will allow the TCP/IP, UDP/IP, and UDP/IP Multicasting protocols to achieve throughput limited by the bus speed of the machine and the speed of the network interface. The main points of the architecture are a fast track for common case packets and an early packet demultiplexer. We wrote code for a sample fast path to test the latency of this architecture. This demonstrated that the architecture provides very low latency to TCP/IP and UDP/IP packets. This latency is an order of magnitude faster than any proposed implementation that we know of, and is comparable to fully customized protocols such as Active Messages. The remaining bottlenecks are in the cost of hardware interrupts at the reciever and in the context switch at the sender. With this architecture, protocols such as RMP will be able to provide fault-tolerant, totally ordered packets to distributed systems in 200 microseconds or less.

References

[ADKM91] Y. Amir, D. Dolev, S. Kramer and D. Malki. "Transis: A Communication Sub-system for High Availability." Technical Report CS9113, Hebrew University of Jerusalem, Nov. 1991.

[AFM92] S. Armstrong, A. Freier, K. Marzullo. "Multicast Transport Protocol". RFC1301. (February, 1992).

[AMSM92] D. A. Agarwal, P. M. Melliar-Smith, and L. E. Moser. "Totem: A protocol for message ordering in a wide-area network." Proceedings of the First ISMM International Conference on Computer Communications and Networks (San Diego, CA, June 1992). pp. 1-5.

[BTS93] C. Brendan, S. Traw, J. Smith. "Hardware/Software Organization of a High-Performance ATM Host Interface." IEEE Journal on Selected Areas in Communications, vol. 11, no. 2. pp. 240-253, February 1993.

[Brust94] J. Brustoloni. "Exposed Buffering and Sub-Datagram Flow Control in ATM LANs." Proceedings of IEEE Local Computer Networks, Minneapolis, MN. pp. 176-185, October, 1994.

[Clark89] D. Clark et. al. "An Analysis of TCP Processing Overhead." IEEE Communications Magazine. vol. 27, no. 6. pp. 23-29, June 1989.

[CoSt91] D. Comer, D. Stevens. Interworking with TCP/IP: Design, Implementation and Internals. Prentice Hall, 1991.

[Dalt93] C. Dalton, et. al. "Afterburner." IEEE Network Magazine. pp. 36-43. July, 1993.

[Davie93] B. Davie. "The Architecture and Implementation of a High-Speed Host Interface." IEEE Journal on Selected Areas of Communications, vol. 11, no. 2. pp. 228-239, February 1993.

[Drus93] P. Druschel et. al. "Network Subsystem Design." IEEE Network Magazine. pp. 8-17. July, 1993.

[DrPe93] P. Druschel, L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility". Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles", December, 1993.

[Jacob92] V. Jacobson, et. al. "TCP Extensions for High Performance." RFC 1323. May, 1992.

[Jacob93] V. Jacobson, "A High-Performance TCP/IP Implementation." Gigabit-per-Second TCP Workshop, Center for National Research Initiatives, Reston Va., March 1993.

[KaPa93] J. Kay and J. Pasquale. "The Importance of Non-Data Touching Processing Overheads in TCP/IP." Proceedings of SIGCOMM '93, Ithaca, N.Y., Sept. 1993.

[Mart94] R. Martin. "HPAM: An Active Message Layer for a Network of HP Workstations." Work in progress.

[Mart2/94] R. Martin. Personal interview.

[MBH94] R. Minnich, D. Burns, F. Hady. "A 1.2 GBit/sec, 1 microsecond latency ATM interface." USENIX High Speed Networking Symposium, Oakland, CA. pp. 73-89, August 1994.

[McCanne94] S. McCanne. Personal interview.

[Mold94] K. Moldeklev, et. al. "TCP/IP Behavior in a High-Speed Local ATM Network Environment." Proceedings of IEEE Local Computer Networks, Minneapolis, MN. pp. 176-185, October, 1994.

[PaPi94] C. Partridge, S. Pink. "A Faster UDP." Work in progress.

[Paxs94] V. Paxson. Personal interview.

[SmTr93] J. Smith, C. Traw. "Giving Applications Access to Gb/s Networking." IEEE Network Magazine. pp. 44-52, July 1993.

[Steve90] W. Stevens. UNIX Network Programming. Prentice Hall, 1990.

[WhMoKa95] B. Whetten, T. Montgomery, S. Kaplan. "A High Performance, Totally Ordered Reliable Multicast Protocol". To be published in Springer-Verlag volume, March, 1995.