BEOWULF LOW LATENCY PROTOCOL

by

Ira Michael Burton

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

March 2003

The project presented by *Ira Michael Burton* entitled *Beowulf Low Latency Protocol* is hereby approved.

Amit Jain, Advisor	Date
Alex Feldman, Committee Member	Date
John Griffin, Committee Member	Date
John R. Pelton, Graduate Dean	Date

ACKNOWLEDGEMENTS

I want to thank my wife, Brooke, for all the patience, help, and encouragement she has given me over the life of this project. I would also like to thank my family for all of the support they have given me, and Amit for everything he has taught me over the last eight years.

ABSTRACT

Beowulf clusters are emerging as a cheap way to harness large amounts of computational power. Transferring data between systems is an expensive part of a parallel algorithm, since the performance of any one node is many times faster than the network's throughput. As network performance improves, the transmission latency becomes a larger percentage of the overall transmission time. Network latency is time lost during a parallel algorithm, leading to performance loss for the overall cluster. This thesis discusses two Linux 2.4.18 kernel patches that reduce the overall runtime of Beowulf applications by reducing latency. This work differs from other high-performance protocol work like U-Net and VIA in that it only requires a one line user-code change to take advantage of the improvement. The performance increase for both protocols is significant, with the fastest protocol running an algorithm using 5 clients in half the time. The kernel patches and test programs can be downloaded at `http://cs.boisestate.edu/~amit/research/lowlatencyTCP`.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Over the last decade, computer network performance has drastically improved. Most of these improvements are in the hardware arena, transmitting messages faster than previous generations. Over the same time, hardware costs have plummeted, opening the doors for multiple fast computers closely connected with fast networks. This has given rise to newer topic in computer science, cluster computing; when tasks that usually run on a single computer are divided up among multiple computers in parallel, the task's runtime is diminished.

While the cost of computers fell and network performance improved, the protocols used to talk between computers have stayed virtually the same. This thesis outlines two new communication protocols that improve the overall cluster performance.

1.1 Background

Before discussing the specifics of the new protocols, some background on the hardware and software behind inter-computer communication is necessary.

1.1.1 Hardware

The hardware inside a modern computer is manufactured by multiple vendors. Each vendor has multiple revisions and versions of each type of hardware. There are standards for hardware interoperability, but not for the software to interface with the hardware. This allows vendors to prescribe an interface that best suites their implementation. A device driver is software that interacts directly with the specifics of hardware, while providing a more generic interface to higher level software components.

One particular type of hardware is a Network Interface Card, or NIC. NICs physically transport messages between computers, converting data into a series of voltages and transmitting them along a wire. This group of signals is a single unit of network communication, commonly called a packet. The receiving computer's NIC converts the packet back into the original message. There are multiple types of networks, but the thesis will be limited to the discussion of Ethernet, since it is the connection mechanism for most computers.

An Ethernet NIC has a cable jack allowing an Ethernet cable to connect systems together. This is a simple way to connect two computers, but it doesn't scale well. It is unreasonable to require 5 NICs just to connect 5 different computers. Additional routing hardware is necessary to overcome this problem. A router receives packets from one machine and routes them to either the destination or another router closer to the destination.

More routers are required as the computers get further apart. Once three routers connect to each other, the network graph becomes cyclic, and as more routers are added, more cycles are created. This leads to multiple paths between computers; the route doesn't matter to the packets as long as they reach their destination in a timely manner.

One advantage of this connection scheme is network redundancy; if one router fails, packets are not sent to that router and the network of computers is not severed. One negative consequence is that if a packet is successfully sent from one computer to the router, there is no guarantee that the packet will reach its destination. This problem has multiple scenarios.

The most obvious problem is when the destination system is not even running, so packet delivery is impossible. Another case is when a packet is corrupted because a router or the transmission medium is not fully functioning. A more complicated example is when multiple packets are sent but take different routes, then it is possible for the first packet to take a slower route than the second. Thus, the packets to arrive out of order. The hardware does handle some packet corruption issues along the way, but it is the software's responsibility to insure integrity.

1.1.2 Software

There are two types of software running on a computer, categorized by their executing location. The kernel-space software is the hardware device drivers and operating system, responsible for abstracting hardware and supplying interfaces to system resources. User-space software runs on top of the operating system using resources, like the file system, to accomplish goals. The operating systems abstracts the network interface so the same user-space programs can run regardless of the NIC's manufacturer.

1.1.3 Beowulf Clusters

A Beowulf cluster is a network of computers that are interconnected, but not necessarily visible to the outside world. Usually the systems are consumer desktop systems, stripped down to only the necessary parts [9]. The goal of a Beowulf is shortening the overall runtime of an algorithm by distributing tasks between multiple machines. The first Beowulf cluster was developed in 1994 for the NASA's Earth and Space Sciences project [10].

One benefit of a cluster is that it is a simple and cheap way to have a lot of computational power. Since a Beowulf is made out of commodity parts, the cost is fairly low. Creating a custom super-computer with the same performance can easily cost ten times as much.

The computers in a Beowulf are physically near each other, which solves some of the issues that software must handle when dealing with networks. The cluster's systems are expected to be running, so every packet has the ability to reach its destination. The packets will not get lost en-route because there is only a single route. This implies that packets do not arrive out of order because there is only one path to take. The Beowulf can be treated as a multiple CPU system, connected via Ethernet instead of a hardware bus.

Linux is the operating system for many Beowulf clusters. One major benefit of Linux is that it is free, so there are no additional software fees associated with adding another box to a cluster. Also, it is open source, so users are able to modify the source for research. Linux has good network performance [4] and has tools already developed to simplify administration of multiple machines. This thesis will focus on developing patches for a modern, stable Linux kernel, namely 2.4.18.

1.2 Goals of the New Protocols

The project's goal is to create a higher-performance communication protocol for Beowulf clusters, taking advantage of the reliability of a local area network. The new protocol will reduce transmission and reception latency, yielding a better performance. The outcome of the project is two kernel patches implementing the new protocol. Two aspects of the new protocol will be tested for performance increases: the transmission latency and the overall throughput. Test code will be designed to focus on these aspects, and the throughput tests should be given more emphasis because they better model real-world software solutions.

1.3 Requirements

Before attempting a solution, there are some requirements that must be fulfilled. First, installing the new protocol cannot break currently running software. Beowulf users will not be likely to try a new solution at the cost of breaking current software. Similarly, the changes must work with current hardware and conform to the current device drivers. Otherwise device driver writers would need to maintain and test twice as much software. Also, the end users cannot be expected to drastically change their coding styles to use the protocol. This means user programs should not require major recoding to take advantage of the new protocol. Since there are volumes of code using TCP/IP, it would be a tremendous project to update the code and retest.

1.3.1 Related Work

There has been substantial previous work in removing the latency from network protocols. Much effort was made to improve Linux network protocols with the release of the 2.4 kernel. A patch is available for the 2.2 kernels to improve performance, but not for 2.4 because of improved networking changes [4]. Since the new cluster protocol patches are for the 2.4.18 kernel, they are already based on the improved network protocol implementations.

One common theme to improve performance is moving the protocol code out of kernel-space and into user-space, known as User-Level Networking (ULN)[1]; a notable implementation is Cornell's U-Net [2], which uses user-space libraries to demultiplex received messages. The main goal behind the ULN model is to achieve a zero-copy protocol, where there are no additional copies made to kernel space, an expensive operation [3]. One of the most promising implementations of this idea is the Virtual Interface Architecture (VIA), supported by Microsoft, Compaq, and Intel [1]. All of these implementations require additional user-level libraries and network device drivers to use the protocols. Not requiring device driver modifications or changing coding styles is the major difference of this project.

1.3.2 Simple Solution

The easiest way to reduce latency is to do less work when sending and receiving. A simple solution is to circumvent all of the current networking code in the kernel. The easiest way to do this is by creating a new Linux device, via a `/dev` entry, that talks directly to the device driver. This would be the best performing solution but introduces more problems than it solves. The biggest difficulty would be sharing the device between applications; that is, how to make two separate applications read from the same device at the same time. One method is to create multiple `/dev` entries, but this would limit the number of connections. Also, it is a software abstraction which seems to be a nearsighted implementation. This solution has another limitation, supporting other network communication and the new cluster protocols simultaneously on the same device, since they share the underlying hardware. How could the device driver know to which path to send data without modifying the device driver? The only way to avoid this problem is to make the change within the socket layer itself.

1.4 Sockets And TCP/IP

A socket is a inter-process communication abstraction created by the operating system for user-space programs. Sockets are connection based, meaning two sockets are connected only to each other, and nothing else. Simple send and receive interfaces hide complicated data transactions; the benefit of using a socket is that both programs are oblivious as to the other's location. Regardless of whether or not the processes run on the same machine, there is no code change. The socket usage in this thesis is limited to communication between systems.

There are multiple types of sockets, designated by family and protocol; the family dictates how the socket should behave, while the protocol determines the specific implementation used by the family. This thesis limits itself to the `AF_INET` family used to communicate via Ethernet. The particular protocol is TCP/IP version 4, the most commonly used Internet protocol. The changes discussed can be ported to IP version 6, which has even more network latency.

The operating system supports multiple active sockets, meaning user programs can connect to many other programs on any number of systems. The operating system manages these sockets preventing user-level programs from knowing that other programs are sharing the hardware. Two user programs can send and receive at the same time without stepping on each other's toes. The operating system insures that programs only receive data from the other end of their socket and not data from another connection. This is not a trivial task because the hardware is shared.

The International Standard Organization (ISO) developed the Open System Interconnect (OSI) model simplifying this complication [5]. Each layer builds on assumptions of the previous layers, breaking the problem into seven pieces, each with its own goal. Every layer adds additional abstraction to the communication process, so high level errors are not seen by the lowest layers and vice versa.

1.4.1 TCP/IP

TCP/IP (Transmission Control Protocol / Internet Protocol) is a socket implementation that connects computers via an unreliable network [7]. The main benefits

Table 1.1 The ISO OSI Seven Layer Model for Networking

Application	The upper most layer, represents data to the user.
Presentation	Verifies syntax and semantics of the data.
Session	Manages control of the protocol.
Transport	Breaks data into acceptable pieces, and builds them up later.
Network	Determines how packets are routed
Datalink	Transmits the packets using hardware.
Physical	Sends data along a physical medium.

are: reliable message delivery, reliable message order, data integrity checking, and the ability to retry failed messages. This protocol solves the problems discussed in section 1.1.1

It is difficult to map all TCP/IP tasks into the OSI seven layer model. This was first discovered with ARPANET, a predecessor to TCP/IP [5]. In order to better model the network interaction, some layers of the OSI stack were removed and refined to become what is known as the TCP/IP Reference Model.

Table 1.2 The TCP/IP Reference Model

Application	The user-level program using the socket.
Transport	Allows source and destinations to carry on conversations.
Internet	Allows hosts to inject packets onto a network
Host-to-Network	Combination of the Physical and Data link layers

The Application layer is in user-space, but the other three are the kernel's responsibility. The TCP code implements the Transport layer, and the IP handles the Internet layer. The device driver is half of the Host-to-Network component, while the NIC firmware covers the other half.

The TCP layer is responsible for queuing data to be sent or received, checksumming messages for data integrity, and ensuring correct message order. The IP layer is responsible for validating message integrity and breaking the message into appropriate pieces. One component of the IP layer is the Maximum Segment Size, or MSS, which determines the largest allowed message at the moment, this acts as a throttle for communication. If a system starts to get behind in handling messages, its MSS starts to drop. If the MSS drops to zero, then no additional messages can be sent to the device. The effective MSS is computed for every message, insuring that the sender and receiver are not getting overwhelmed.

Another component of TCP/IP is the message window; the window is the amount of data that can be sent before an acknowledgment from the receiver. The windowing information is sent as part of every message and is used to compute the MSS. The window and MSS both control message bandwidth, but solve independent problems. The window determines how often sender and receiver to synchronize, while the MSS determines how much data the receiver can store due to memory limitations.

1.4.2 Packets

A packet is a structure used to encapsulate data specific to each networking layer. At the core of the packet is the original message, but as the packet passes through additional software layers, each layer wraps the packet with additional information. The goal behind this is to have each layer treat the inner layer's data as the original message. Then when the packet is received, the layers peel off their outer wrappings in the reverse order, leaving the original message intact.

Assume a process is sending a 10 byte message. The message is first handed to the TCP layer.

Table 1.3 Message before Reaching the TCP Layer

Original Message	10 bytes

At the TCP layer, a TCP header is appended to the start of the packet. The messages length, checksum, and other options fill the header.

Table 1.4 Message after the TCP Layer

TCP Header	At least 20 bytes
Original Message	10 bytes

Once the TCP packet is created, it is handed to the IP layer, which wraps the data with its header, containing source and destination addresses, overall packet size, and many option flags.

Table 1.5 Message after the IP Layer

IP Header	At least 20 bytes
TCP Header	At least 20 bytes
Original Message	10 bytes

At this point the packet is ready for transmission, and much bigger than the original message. All of this header information makes TCP/IP reliable on an unreliable network.

Chapter 2

IMPLEMENTING SOLUTIONS

The robustness of TCP/IP is excessive for the Beowulf's reliable network since a significant amount of kernel time is spent doing tasks like computing checksums for messages. One common distributed computing paradigm is a client-server architecture, where servers send commands to clients. When a client finishes its task, it requests more work, requiring the server to send another task. This makes the server a bottleneck, so if the server is doing something unnecessary, like building a checksum, then clients may be sitting idle. Reducing the number of unneeded actions done by the clients and servers improves the overall cluster performance. Two new protocols, using TCP/IP as a basis, were created, with the slower one called Cluster B, and the faster Cluster A.

Since the implementation requires a kernel change, not creating bugs is crucial. Users will not install the software, no matter the performance gain, at the cost of instability. The primary implementation detail is to minimize the number of differing lines between the cluster kernel and the stock kernel. Code is modified using these rules:

1. If existing code can be used without modification, then use it.

2. Otherwise, modify as little as possible.

3. If new code is required, mirror the actions taken by the existing code, simplifying future maintenance.

Many routers and NICs are optimized for transmitting TCP/IP packets. Since the Maximum Transmission Unit, or MTU, for the hardware is usually 1500 bytes, we cannot raise this amount without putting the end users in jeopardy.

The 2.4.18 Linux kernel uses a structure called an SKB, see Appendix C for details, to transmit data to and from a network device driver. The SKB structure is not specific to TCP/IP, but any change would require modifications to all network drivers, violating the requirements in Section 1.3.

2.1 TCP/IP Protocol

Users pass the family, type, and protocol when creating a socket which initializes the socket, so calls beyond that point reference the correct implementation. The example below is code that creates a TCP/IP socket.

```
sock = socket(AF_INET, SOCK_STREAM, IPP_PROTOTCP)
```

This function creates a socket and returns a descriptor for the socket. The socket's family is `AF_INET`. The `SOCK_STREAM` defines how the socket communicates, and `IPP_PROTOTCP` is the TCP/IP protocol. After making the call, the socket descriptor is stored by the operating system.

Since user-level code uses sockets to connect two process together, each process must declare its own socket. The server listens and waits for the client to connect, because of this, the server makes additional preparations for the client. First, `bind()` must be called, associating the server's IP address and the port to the socket. This prevents other processes from using that same port. Then, the server calls `listen()`, allowing it to listen for client connections. Lastly, the server calls `accept()`, and waits for a client program to connect.

The client program creates a socket, and connects to the server, using the server's IP address and port. If the server is not listening, then the client fails to connect. Once the client's connection is accepted, then the two processes can communicate. Processes send message via `send()`, and receive with `recv()` (or read/write). The messages are queued up by TCP layer on both sides of the wire, so that each process can transmit and receive simultaneously without interfering with each other.

2.1.1 Sending

When a program makes a send request, it passes the socket descriptor, a pointer to the message, and the number of bytes to send. A flag is passed to tell how the socket behaves, for instance if the `send()` call should send a signal on error or bypass a gateway. This thesis assumes that no flag is passed in.

```
send(sock, (const void*)msg, numBytes, 0)
```

The socket descriptor is looked up in a list of active descriptors, and the send request is passed to the family's send function. The `AF_INET` send algorithm verifies that the socket is setup correctly and calls the protocol's send function. The algorithm below shows the steps taken in the TCP/IP `send()` routine.

1. Copy the message from user-space to kernel-space.

   (a) If there is an old SKB available, use it. Otherwise create a new SKB.

   (b) If the internal SKB buffer can hold the entire message, then copy the new message into the SKB, and perform a checksum.

   (c) If all of the data cannot be copied, see if the SKB can be expanded to fit all of the data. If so, expand the SKB, copy the message, and compute a checksum.

   (d) Otherwise, copy what we can into the current SKB, assuming more room will be available in the future.

2. Queuing the SKB

   (a) If the send queue is empty, put the SKB in the queue and build the IP packet.

   (b) Otherwise, put this SKB at the end of the queue, and attempt to transmit the head of the queue. If it transmits, then build the IP packet for the new head in the SKB queue.

3. Building the IP packet

   (a) Check the effective MSS, and if there is enough room for the entire packet, then build the IP packet. Otherwise, split up the SKB packet.

   (b) Add the IP data to the SKB structure.

   (c) Determine the destination of the packet.

(d) Compute a checksum for the IP packet.

(e) Modify the MSS and window.

4. Determine if the packet should be sent, filter out packets sent to systems that are being ignored.

5. Hand the SKB to the IP layer, which queues the packet to be sent to the device driver.

6. If the MSS has not fallen to zero, and there are still SKBs in the send queue, go back to Step 3.

7. If there is still data to be sent, but still no memory, then wait for more memory, otherwise go to Step 1.

The time it takes the user program to switch to kernel-level, lookup the socket, hand the data to the correct protocol, run this algorithm, and send the packet via the device driver is the *transmission latency*.

2.1.2 Receiving

When a program makes a receive request, it passes the socket descriptor, the array of bytes to be filled, and the number of bytes it is expecting to read. A flag is passed to tell how the socket behaves; this thesis assumes that sockets will read all of the requested bytes before returning.

```
recv(sock, (void*)msg, numBytes, MSG_WAITALL)
```

Once again the socket descriptor is looked up and the request passed off to the family and the protocol. A program can request to read a packet before, during, and after the transmitting program sends the information. The receiver waits until all of the data is read or there is an error in the process.

Messages are received asynchronously via the device driver; there is nothing the user-level code does to accept packets. When a user program reads data from a socket, it is reading from the receive queue. A receiver can get behind in reading, but is guaranteed to get the proper order when the data is read. The TCP/IP `recv()` algorithm is below.

1. Check the receive queue.
   (a) See how many more bytes are required. If none, then return the number of bytes read.
   (b) If the receive queue is empty and more data is required, then go to Step 2.
   (c) If the top SKB in the receive queue is larger than the amount left to be read, then read some of the message.
      i. Read the required number of bytes out of the SKB.
      ii. Shrink the SKB to reflect the amount read.
      iii. Leave it as the head of the queue.
      iv. Return total number of bytes read.
   (d) Else, the top SKB contains some of the message.
      i. Read the entire SKB.

ii. Update the number of bytes still required.

iii. Remove this SKB from the receive queue and go back to Step 1.

2. Check to see if the packet was in prequeue, meaning it was received but is not in the receive queue yet. If so, then wait for the prequeue to finish and go back to Step 1.

3. More data is still required but has not yet arrived.

   (a) Wait for the receive queue to change size.

   (b) Once it has, go back to Step 1.

This algorithm makes it difficult to time the actual `recv()` latency. For example, assume a server is going to send a client two messages, one is 1000 bytes long and the other 1 byte long. If the 1000 byte message is sent well before the client's `recv()` call, then the transmitted message would already be in the receive queue. When the client calls `recv()`, it would be artificially fast. Likewise, if the client calls `recv()` before the server sends the 1 byte packet, the client would have to wait for the transmission of 1 byte. This makes the 1000 byte message appear an order of magnitude faster than a 1 byte message.

The `recv()` algorithm requires the use of the receive queue, which is filled asynchronously. The receive queue is not filled until a packet is verified that it is in the correct order via the prequeue. The `driver_recv()` algorithm below is triggered by the device driver, it fills the receive queue, determines message order, and alerts blocking `recv()` calls.

1. The data is read via the NIC.
2. The NIC's device driver builds an SKB, then hands it to the IP handler.
3. The IP header is validated, determining if malformed malicious packets are being sent. Bad packets are properly handled, usually via ignoring the packet.
4. The IP packet's checksum is validated, and if the packet is mangled, a retransmission request is sent.
5. The sender's IP address is filtered, and the packet is ignored if it is from an unwanted system.
6. The SKB is passed to the prequeue process.
   (a) The TCP packet's checksum is validated, and the MSS is updated.
   (b) The packet is placed in the prequeue queue, and the packet order is verified.
   (c) If the packet is out of order, then a request is made to resend the missing packet.
7. Once the prequeue packets are in order, a new SKB is created, the data copied, and placed into the receive queue.

TCP also sends additional messages besides the messages carrying data. One of these messages is an acknowledgment, or ACK. The ACK is a notification that messages were safely transmitted which forces a MSS update, the lack of an ACK means that a message may have been lost, and a resend occurs.

When the IP layer transmits a message, the actual message is retained well after the sender has completed the function. The transmitted messages are stored until the receiver sends the ACK. Once it is received, the original sender removes the SKB from memory and updates its send window. The ACKs are difficult to time because they happen asynchronously and have no corresponding calls from user-space.

2.2 Cluster B Implementation

The first implementation of a lower latency protocol removes the TCP and IP checksums and options from the packet creation process. A new socket type was created in the `AF_INET` family supporting the new cluster protocol, `SOCK_CLUSTER`. The only changes to the generic socket code was the modification of the `AF_INET` family to accept socket communication of `SOCK_CLUSTER`. A module must support a particular set of functions for proper interaction with the `AF_INET` family, two of these are send and receive. Since the goal of the project is to speed up those two functions and minimize the amount of other code change, many of the TCP/IP methods are reused. This greatly reduces the amount of new code, and binds the cluster interface closer to the current TCP/IP implementation.

Because of this implementation, it is trivial to port a program using TCP/IP to one using `SOCK_CLUSTER`. The only line of code that changes is socket creation.

```
sock = socket(AF_INET, SOCK_CLUSTER, 0)
```

No protocol is passed in, since the `SOCK_CLUSTER` type has only one protocol. The first implementation of the cluster code focuses on modifying the TCP/IP sending loop in Section 2.1.1. The checksumming is removed from the TCP and IP layers on both transmission and reception. Also, the network filtering code is no longer executed, used to ignore messages from particular machines usually as a reaction to malicious packets. Many of the special case paths for TCP/IP options are no longer supported. In the end, it is a streamlined version of TCP/IP code.

2.2.1 Sending

The sending algorithm is nearly identical to TCP/IP.

1. Copy the message from user-space to kernel-space.

   (a) If there is an old SKB available, use it. Otherwise create a new SKB. (*Identical to original TCP/IP algorithm*)

   (b) If the internal SKB buffer can hold the entire message, then copy the new message into the SKB. (*No longer performs checksum*)

   (c) Since all of the data cannot be copied, see if the SKB can be expanded to fit all of the data. If so, expand the SKB and copy the message (*No longer performs checksum*).

   (d) Copy as much as possible into the current SKB, assuming more room will be available in the future. (*Identical*)

2. Queuing the SKB (*Identical*)

   (a) If the send queue is empty, put the SKB in the queue and build the IP packet.

   (b) Otherwise, put this SKB at the end of the queue, and attempt to transmit the head of the queue. If it transmits, then build the IP packet for the new head in the SKB queue.

3. Building the IP packet (*No checksum computed for the IP packet, otherwise identical*)

   (a) Check the effective MSS, and if there is enough room for the entire packet, then build the IP packet. Otherwise, split up the SKB packet.

   (b) Add the IP data to the SKB structure.

   (c) Determine the destination of the packet.

   (d) Modify the MSS and window.

4. Hand the SKB to the IP layer, which queues the packet to be sent to the device driver. (*Identical*)

5. If the MSS has not fallen to zero, and there are still SKBs in the send queue, go back to Step 4. (*Identical*)

6. If there is still data to be sent, but still no memory, then wait for more memory, otherwise go to Step 1. (*Identical*)

2.2.2 Receiving

Likewise, receiving data is nearly identical to the TCP/IP mechanisms. The Cluster B `recv()` code is identical to the TCP/IP code, so no speed improvement is made. There were minor changes to the `driver_recv()` algorithm.

1. The data is read via the NIC. (*Identical*)
2. The NIC's driver builds an SKB, and hands it to the IP handler. (*Identical*)
3. (*The Cluster IP header is not validated nor is the IP packet's checksum. All network filtering code is also removed.*)
4. The SKB is passed to the prequeue process, which determines if this packet is in the correct order, and updates the MSS. (*No TCP/IP checksum validation*)
5. Once the prequeue is done, a new SKB is created, the data copied, and is placed into the receive queue. (*Identical*)

The ACK and windowing code must remain intact because the current send and receive buffers are still in use. The Cluster A kernel achieves an even smaller software overhead by removing the need for ACK and windowing code.

2.3 Cluster A Implementation

The second implementation abandons much of the original TCP/IP code to gain more speed. This implementation still requires storing data beyond the life of the user-space call, not for the possibility of a retransmit, but for the network device driver to use.

Once a message is queued for transmission, the device driver is signaled and eventually task-switched in to copy the data to the NIC's hardware. The kernel code does not block, waiting for the device driver, but must take it into account. If the data being sent is modified before the device driver gets to it, then the modified data is sent, not the original. To prevent message corruption and unnecessarily slowing down the kernel, multiple SKBs are used. This way, the device driver may operate on one SKB, while the kernel updates the next. At protocol creation time, a ring of MTU sized SKB buffers are created and linked, so after a message is signaled for transmission, the next buffer in the ring is used. There is some additional cost at socket startup compared to the Cluster B implementation, but that only happens once for the lifetime of the socket. The ring's length is a kernel parameter, and has a default value of 7. If a message requires more than seven packets to complete, then the ring buffer fills up and must wait on the device driver, and the overrun count is incremented. The size of the ring is modifiable at runtime, via `sysctl` or writing to the `/proc` file system, and can easily be expanded to prevent any buffer overruns. A values of 7 was used to keep the kernel space requirements low, but it is assumed end-users will tune the size for their needs. When the socket is closed, the overrun count is printed, but only if an overrun occurred.

2.3.1 Sending

The sending algorithm is greatly simplified using the SKB ring. All of the code determining if packets should be split is gone, as is the distinction between the TCP and IP layers. As mentioned in Section 1.4.2, each networking layer adds additional data to the packet. Since this protocol knows how the data is being sent and is storing a ring of reusable SKBs, some optimizations were made. The SKBs are created in a nearly transmittable form. The only thing missing is the actual message payload.

Table 2.1 Ring Buffer SKB

Pseudo-IP Header (Needs length fix-up)	20 bytes
Pseudo-TCP Header (Needs length fix-up)	20 bytes
Message Placeholder	1428 bytes

The TCP and IP pseudo-headers are the minimum header size supported by the Linux TCP/IP implementation. During the packet transmission process, the device driver adds data to the head of the packet. All of the header information combined takes up 72 bytes, leaving only 1428 bytes for the payload. If the payload was any larger, the message would be more than 1500 bytes.

Once a message comes in, the data is copied to the Message Placeholder field, and the correct length fields are fixed up. The fix-up is trivial because there is no checksumming, and all of the header lengths are known in advance because no TCP nor IP options are supported. Since packets arrive in order, the TCP sequence numbers are not updated, requiring very little work to prepare a packet for transmission, simplifying the `send()` algorithm.

1. Copy the message from user-space to kernel-space.

   (a) Get the next SKB in the ring

   (b) If the SKB is being used by the device driver, then mark a SKB overrun and wait a microsecond. Then check again, and wait if necessary.

2. Copy as much data as possible, up to a MTU into the SKB.

3. Update the SKB structure to reflect its new size.

4. Pass the SKB to the IP layer.

5. Get the route table info and hand to the device driver.

6. Go back to Step 1, if there is still data left to send.

This differs greatly from the TCP model, since the SKBs are allocated and configured beforehand. All of the checksum code, window updating, and queuing is missing from Cluster A's implementation.

2.3.2 Receiving

Since the packets arrive in order, there is no prequeue step, this simplifies the implementation. If a packet is not in the receive queue, the protocol will wait until one arrives. Below is the Cluster A's `recv()` algorithm.

1. Check the receive queue.

   (a) If the receive queue is empty, wait.

2. Get the SKB at the head of the queue.

3. Read data out of the SKB

   (a) If the request amount is larger than the total data in the SKB, then read all of it and delete the SKB.

   (b) Otherwise, read as much as possible.

4. If more data is required, then go back to Step 1.

Again, packets do not arrive out of order, so there is no sorting step in the asynchronous `driver_recv()` call.

1. The data is read via the NIC.

2. The NICs driver builds an SKB, and hands it to the Cluster A handler.

3. Packet is placed in the receive queue. (*Nothing is validated*)

This is close to optimal for queuing packets, it simply copies the packet to kernel-space from the NIC.

2.4 Protocol Comparisons

Both of the proposed solutions reuse ideas and structures from TCP, but differ in the algorithms. Both Cluster B and A do not support packet filtering since it is intended for a secure network. Clusters with connection to the outside world should keep this in mind though. This is acceptable though, because the protocol is meant for secure isolated systems.

Both the TCP/IP and Cluster B return from the transmission calls once the packet is in the send queue, but not the device driver's send queue. This allows additional send calls to place data in the first packet, before it is sent out. The Cluster A `send()` call doesn't return until the packet is queued up for transmission, so the packet cannot be modified by additional requests. The TCP/IP and Cluster B protocols require more messages than the Cluster A to pass the same amount of data, unless the situation is right for TCP/IP to combine packets.

2.4.1 Packet Combining

Since TCP/IP and Cluster B keep a transmission queue, additional `send()` requests may occur before the packet is transmitted. This allows updating the queued packet with another message, basically sending two messages in the same packet, space permitting. These protocols have an advantage when sending multiple small messages rapidly.

Chapter 3

PROTOCOL PERFORMANCE

There are two main test cases to determine the performance of the protocol. The first is a transmission rate test, determining the sending latency of the protocol. In this test, a series of bytes is sent to a client over and over without interruption. The time is measured from the first send till the last send.

The second is a throughput test, measuring round trip time. The server sends a message to the client and the client receives it. Then the client sends a one byte acknowledgment message back to the server, while the server waits for the response. Once the response comes back, a single round trip message is complete. This test also works for multiple systems where the server sends a message to each client, then waits for a response from the first client. Once that response arrives, the server waits for a response from the second, and so on. Once all clients respond, one round trip is made.

3.1 TCP/IP Comparisons

Since both protocols share a common protocol name, `SOCK_CLUSTER`, the same kernel cannot support both Cluster A and Cluster B protocols simultaneously. So two kernels are required to test the new protocols. The unpatched kernel is considered the stock kernel. The patched kernel supporting Cluster B is called kernel B, and likewise kernel A supports Cluster A.

One of the original requirements was to not break current software. If these kernel modification degrade the current TCP/IP performance, then this requirement is not met. The same two performance test are used to verify that TCP/IP performance is unchanged. The first is TCP/IP latency, making sure the addition of a new protocol doesn't slow down the process of sending messages. The second is to insure that the overall throughput of TCP/IP is unaffected. All tests are run between two systems using TCP/IP under the stock kernel, kernel B, and kernel A.

3.1.1 TCP/IP Latency Verification

By adding a new protocol, the TCP/IP code path for sending and receiving changes slightly. There is actually only one line of new code from TCP/IP's vantage point. The test cases are: a small packet, a larger packet, nearly a MTU, and messages larger than a MTU. If TCP/IP latency is degrading, it should be evident in one of these tests.

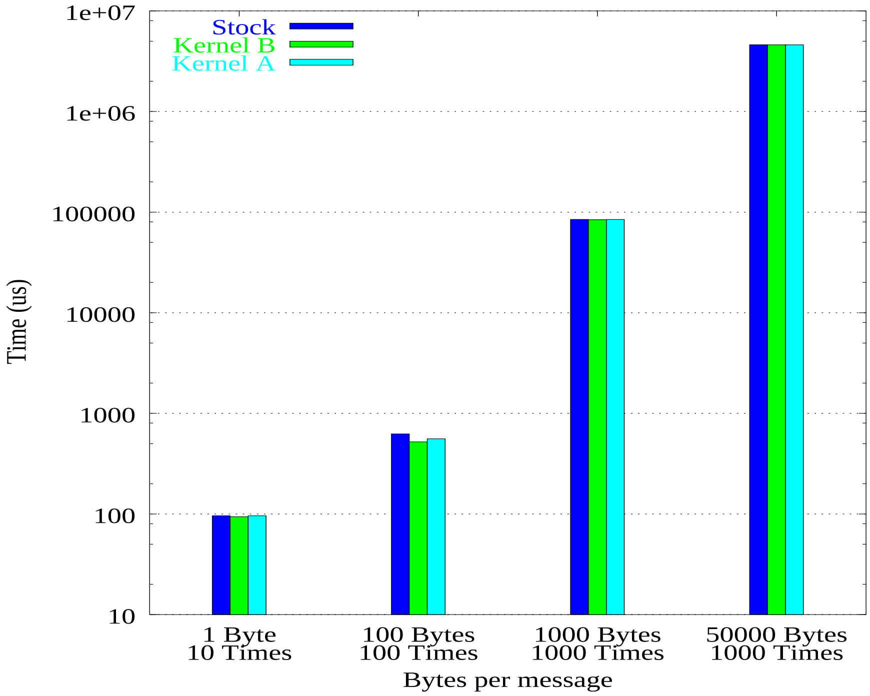

Figure 3.1. TCP/IP Latency Verification

From Figure 3.1, is no evidence that adding a protocol affects TCP/IP latency. In fact both modified kernels were consistently quicker when sending the 100 byte message. The reason for this has not yet been determined.

3.1.2 TCP/IP Performance Verification

Figure 3.2 is the throughput test, also uses a series of different size messages: of a single byte, a few bytes, about an MTU, and some consisting of multiple MTUs. If the transmission queuing and reception code is adversely affected by the additional cluster code, then it will be evident in one of these cases.

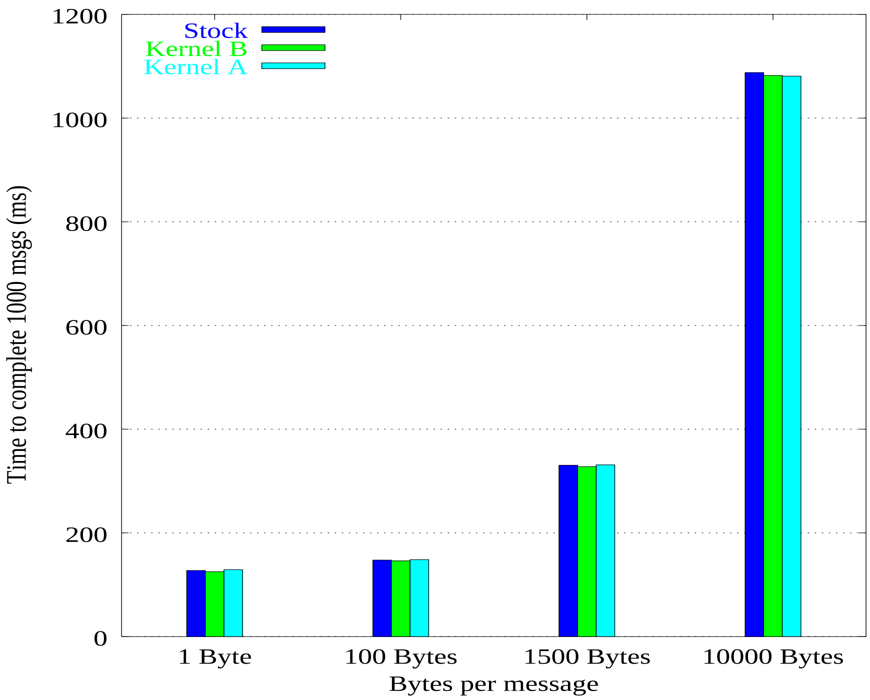

Figure 3.2. TCP/IP Throughput Verification

The TCP/IP code is virtually unaffected by the additional protocol but the performance between kernels is not as identical as the latency test. This is understandable because this test includes the time to send the data across the wire, so there are more uncontrolled variables.

3.2 Small Cluster Performance

These tests were run between two systems, connected directly to each other via a switch. See Appendix A for testing specifics.

3.2.1 Latencies

Since all of the performance gains were done by reducing latency, this test is significant to the success of the project. In all test cases, the TCP/IP test was ran on the stock kernel, the Cluster B test with kernel B, and Cluster A with kernel A.

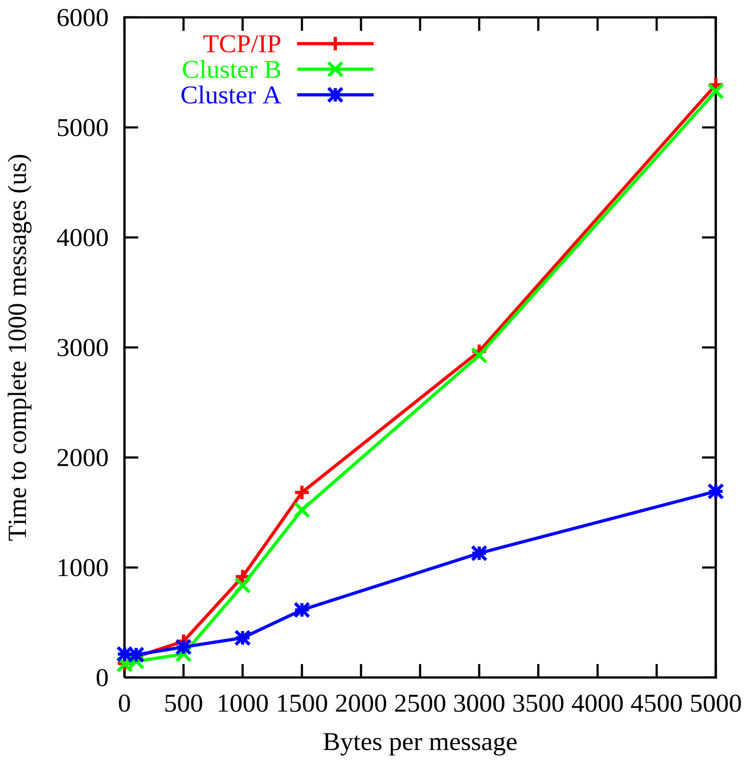

Figure 3.3. Latency Comparisons

There are three significant details in Figure 3.3. The most noticeable is Cluster A's latency improvement. The Cluster A protocol is able to send a 5000 byte message faster than TCP/IP can send a 1500 byte message.

The second detail is how closely the TCP/IP and Cluster B protocols compare. The slopes of these two protocols are basically the same; the Cluster B protocol is just a little bit better. This is exactly as expected because the algorithm for those protocols are the same. Also, after a MTU, the differences between these two protocols is negligible because the time gained by not computing a checksum for the next packet is hidden because the system waits on the device driver.

The third observation is the slopes of the lines. Three main slopes comprise a single line: one between 1 and 1500 bytes, 1500 and 3000 bytes, and above 3000 bytes. These are significant because the first range is just sending a single packet, with or without the packet combining. The second range requires two packets to be sent, making packet combining less significant. The third requires multiple packets to transmit the message. It is expected to see the slope of the line increase for each case due to the additional amount of work. Cluster A's slope is always less than TCP/IP and it increases at a slower rate than TCP/IP as the message size increases.

As mentioned in Section 2.4.1, Cluster B and TCP/IP support packet combining, giving them an advantage in this test when the packets are small. Even with their advantage, the latency improvements of Cluster A keeps the performance for small packets competitive. As the packet size increases, packet merging is less beneficial and the latency improvements of Cluster A become apparent.

3.2.2 Throughput

Throughput is usually the metric for determining performance increase. In this case, it is measured by the time taken to do a series of round trip messages. The only difference between TCP/IP and Cluster B in this test is the transmission and reception latency. These graphs are split into two due to the behavior differences around a MTU length message.

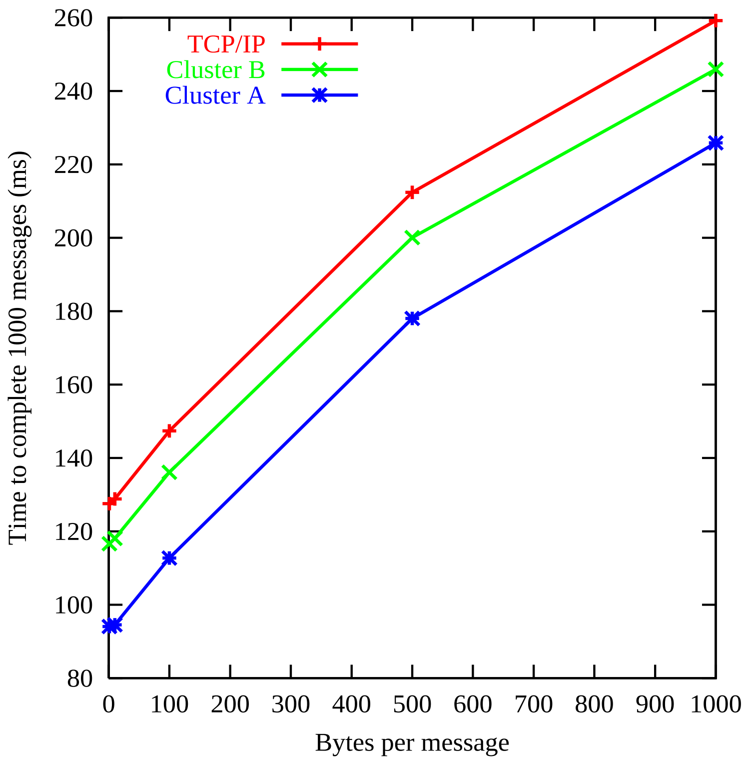

Figure 3.4. Protocol Throughput Comparison Less than a MTU

It's interesting that all three lines in Figure 3.4 are vertical shifts of one another. The sending and receiving latency improvements give Cluster B a 10% speed increase across all packet sizes compared to TCP/IP. It's also noticeable that the algorithm change of Cluster A does not affect the overall slope when talking with a single client, but it does allow for a 20 to 30 percent drop in overall runtime.

As messages get larger, the latency plays a smaller role in the overall message throughput, as shown in Figure 3.5.

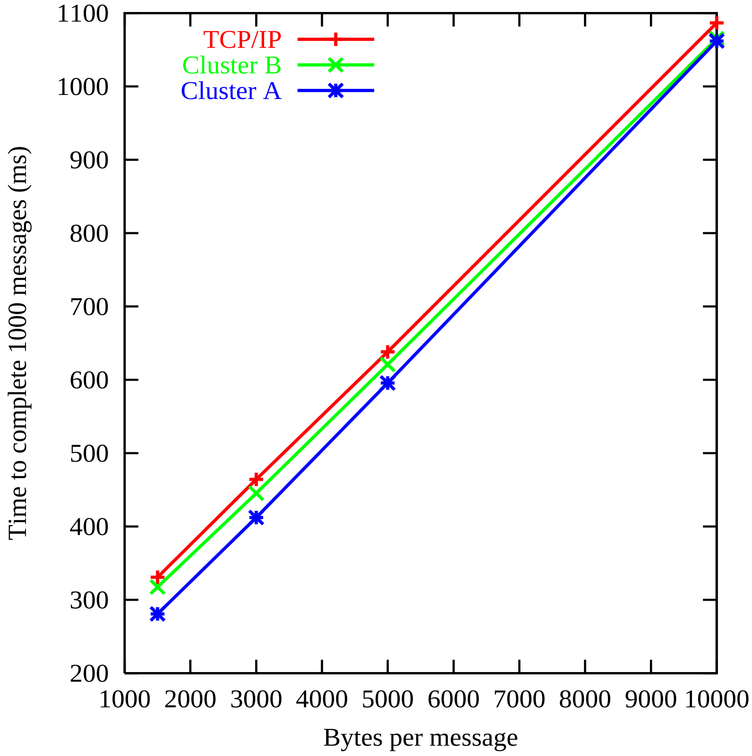

Figure 3.5. Protocol Throughput Comparison Greater than a MTU

While TCP/IP and Cluster B begin to merge after a MTU, Cluster A falls to a 10% improvement. As the message size approaches 10000 bytes, the benefit becomes even more negligible, because the network has reached a saturation point and there is not a significant benefit for getting packets on the wire sooner.

3.3 Larger Cluster Performance

These tests were ran on a larger Beowulf of six systems, see Appendix B for system details.. One system acts as a server with up to 5 clients. These systems are less powerful than the ones from the previous test, so some of the message dynamics are expected to change because the computational time is slower.

One of the biggest benefits of reducing latency is to free up the system so that it can do other tasks. In these tests, the additional tasks may be sending messages to other systems. Assuming a cluster of three systems, it is optimal having three tasks running in parallel. If at anytime a client is waiting on the server, the overall system is not being used to its potential. Reducing the communication latency reduces the probability that the Beowulf is underutilized.

3.3.1 1 Client

Figure 3.6 is a baseline for the rest of the tests. Since these systems are slower than those used on the small cluster tests, the latency is a larger amount of the overall time. This plot shows that as messages get larger than a MTU, the three graphs look less convergent, differing from previous results. This is attributed to the first cluster being fast enough that the transmission time is the bottleneck of the communication.

Due to the implementation of the Cluster A protocol, the difference between a 1428 and a 1429 byte message is substantial. When a message is 1429 bytes, Cluster A sends two packets, resulting in the steeply sloped line between those points. Since the packets are queued by the TCP/IP and Cluster B protocols, sending additional

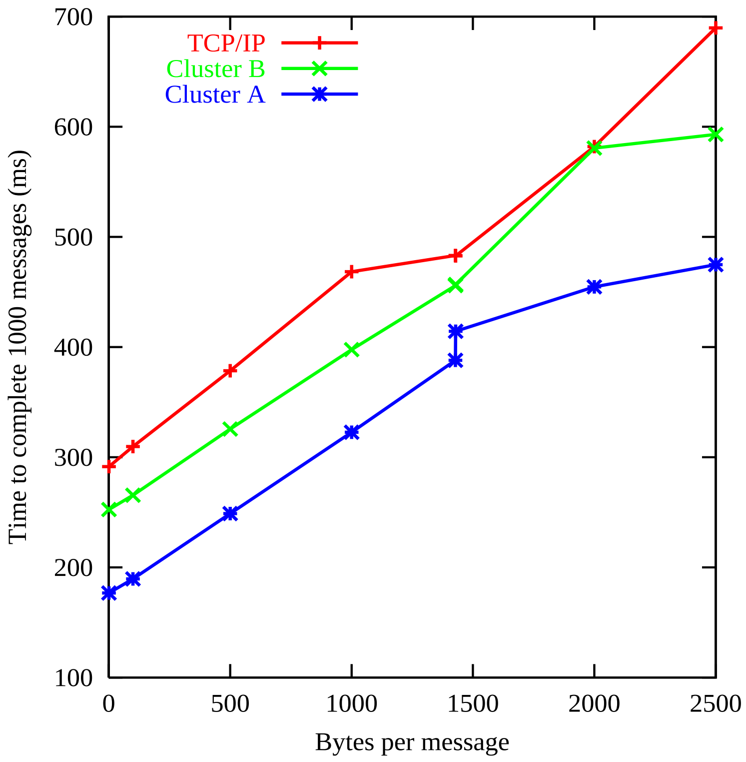

Figure 3.6. Performance with 1 Client

packets is not as discontinuous.

Focusing on Cluster A, the slope of the line drops off after the 1429 byte message. This is due to the device driver sending the first part of the message while the second part is being built. The second part of the message is built quick enough that the device driver can send it right after it finishes the first message. The other protocols have additional overhead forcing the device driver to be called a second time to send the second message.

3.3.2 2 Clients

Figure 3.7 involves two clients. Again, there is a noticeable slope change for Cluster A around a MTU.

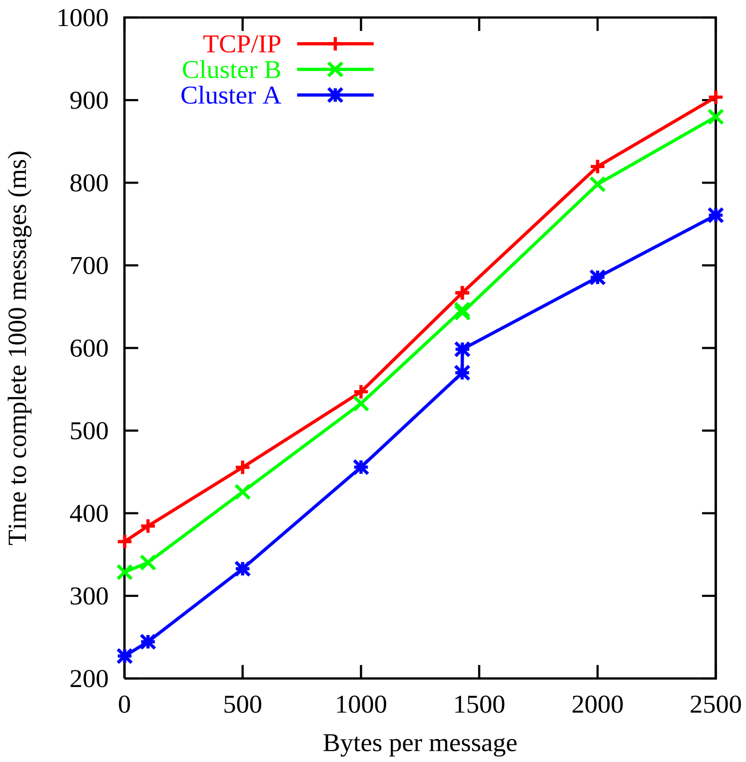

Figure 3.7. Performance with 2 Clients

The Cluster A protocol is always faster than the other two. The difference is greatest when the packet size is the smallest. When transmitting 1 byte, the Cluster A protocol is nearly 40% faster than TCP/IP.

3.3.3 3 Clients

The test forming Figure 3.8 is similar to the previous one except a third client is involved.

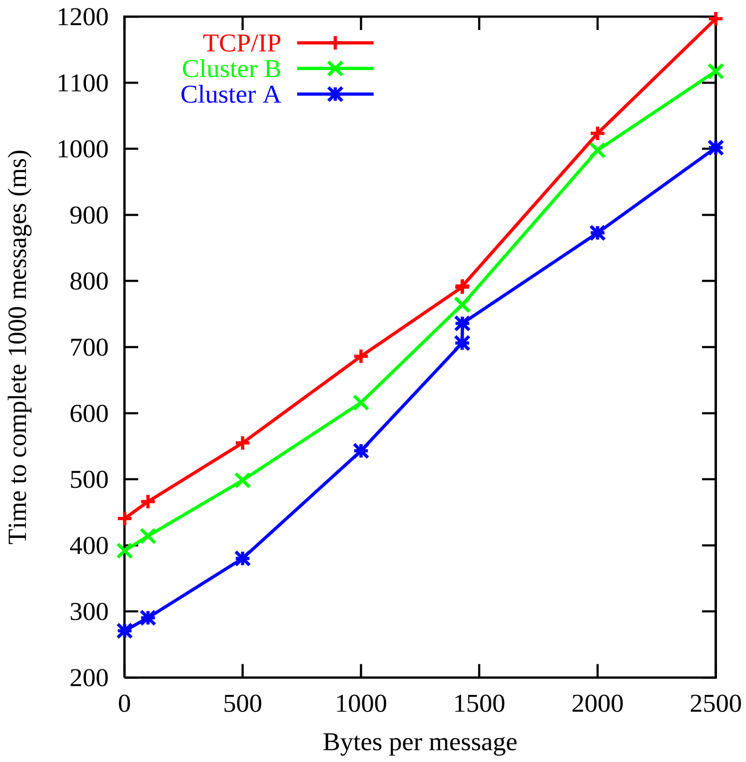

Figure 3.8. Performance with 3 Clients

The results for this test are nearly identical with the previous one. All three lines are basically a linear shift of the previous graph. One interesting property of this graph is the pinching that takes place at a MTU. This is caused by having to wait on the device driver to send the packets, creating a bottleneck. This effect is not very noticeable with the TCP/IP or Cluster B protocol, but Cluster A's performance is hurt having to wait for the device driver.

3.3.4 4 Clients

With the addition of another client, the plots begin to change.

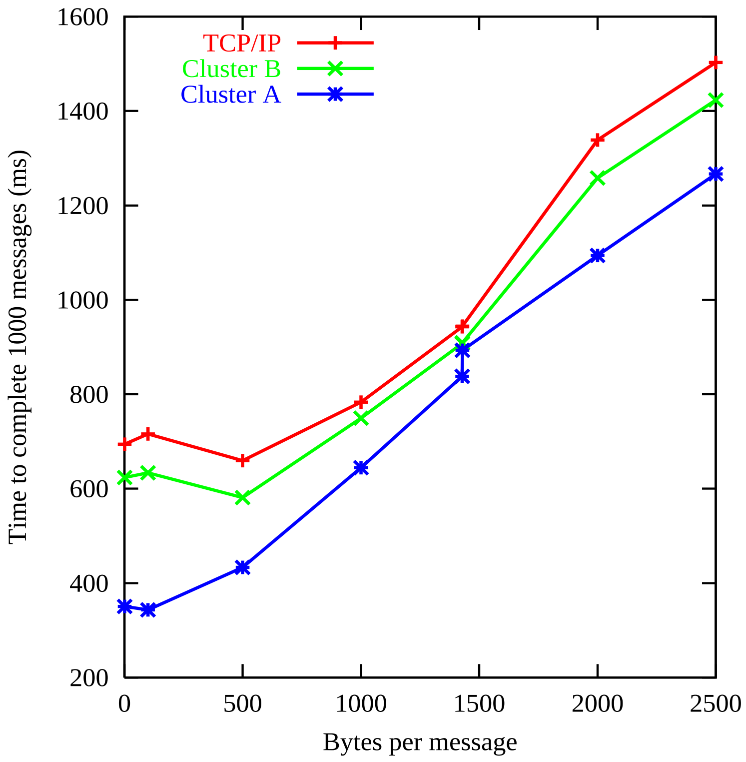

Figure 3.9. Performance with 4 Clients

The TCP/IP and Cluster B protocols behave artificially slower for small packets, which could be due to network overload caused by packet collisions, a consequence of the additional ACK packets. The pinching is even more noticeable where all three protocols perform nearly the same at 1429 bytes. Since two packets need to be sent out, Cluster A can quickly produce the second packet, but then waits. The TCP/IP and Cluster B protocols are not necessarily waiting on the device driver, but are working until the device driver sends the queued SKBs. This allows those protocols to get additional work done for free, where as the Cluster A protocol is idle.

3.3.5 5 Clients

Another client is added making the network overloading even worse.

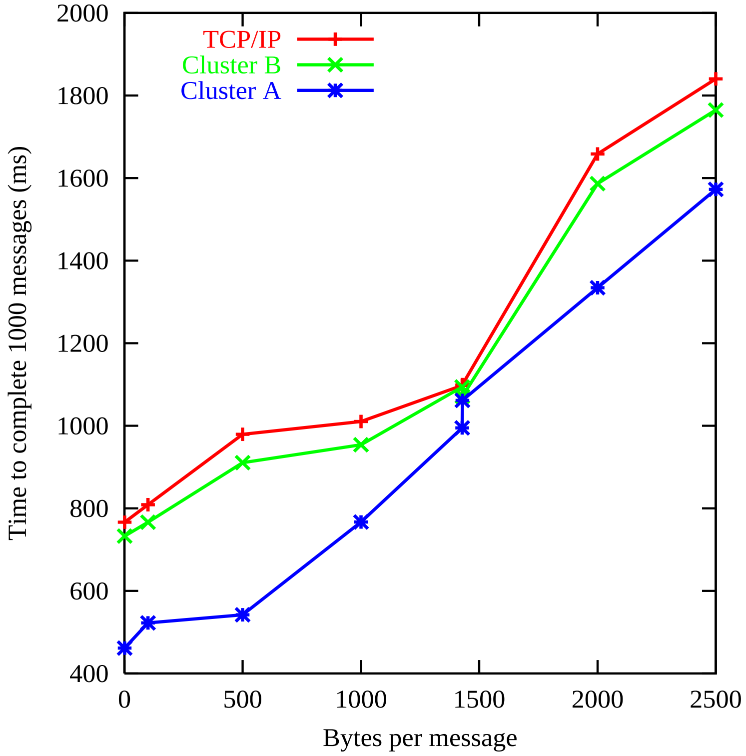

Figure 3.10. Performance with 5 Clients

The performance lost to the network overload makes the 500 byte message look abnormally slow. The pinching has reached a point at which all protocols are virtually the same for a 1429 byte message. Looking at the 500 byte message, the Cluster A protocol finishes the test in half of TCP/IP's time.

3.3.6 Protocol Growth Rates

As the number of clients increase, the overall load of the server system drastically increases. The server not only needs to send out more total messages, but the asynchronous message also increase. The possibility of a client's response being sent at the same time that the server is trying to send to another client is higher, causing more task swapping on the server. In the TCP/IP and Cluster B protocols, the number of ACKs increase for each client, so the overall message count grows. These are plots from the previous graphs, showing the change in message time with respect to client count.

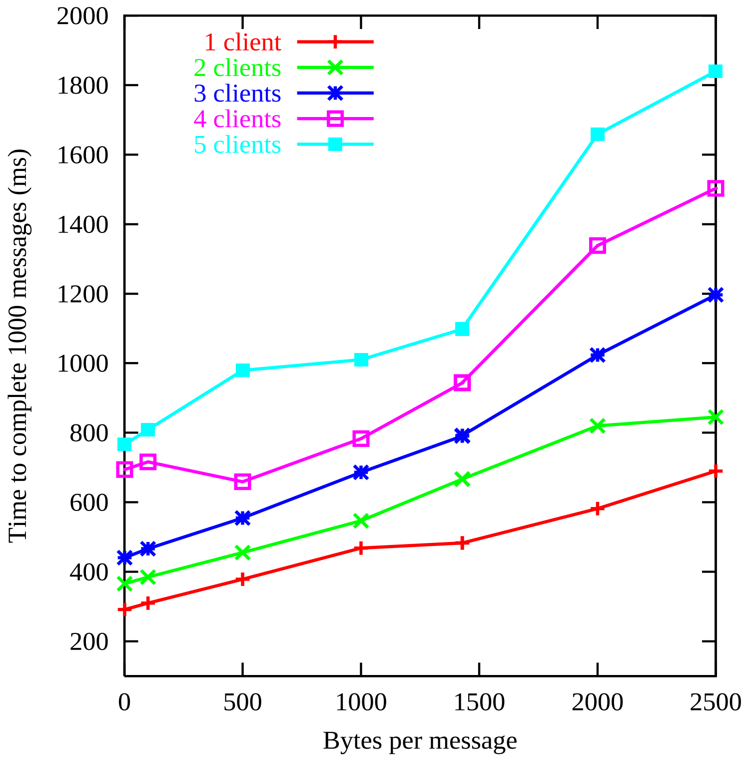

Figure 3.11. TCP/IP Growth

The slopes in Figure 3.11 look fairly linear as long as only three clients are involved. Once the fourth client is introduced, the linear properties are less evident, especially when small messages are sent.

Figure 3.12 contains the same chart but for the Cluster B protocol.

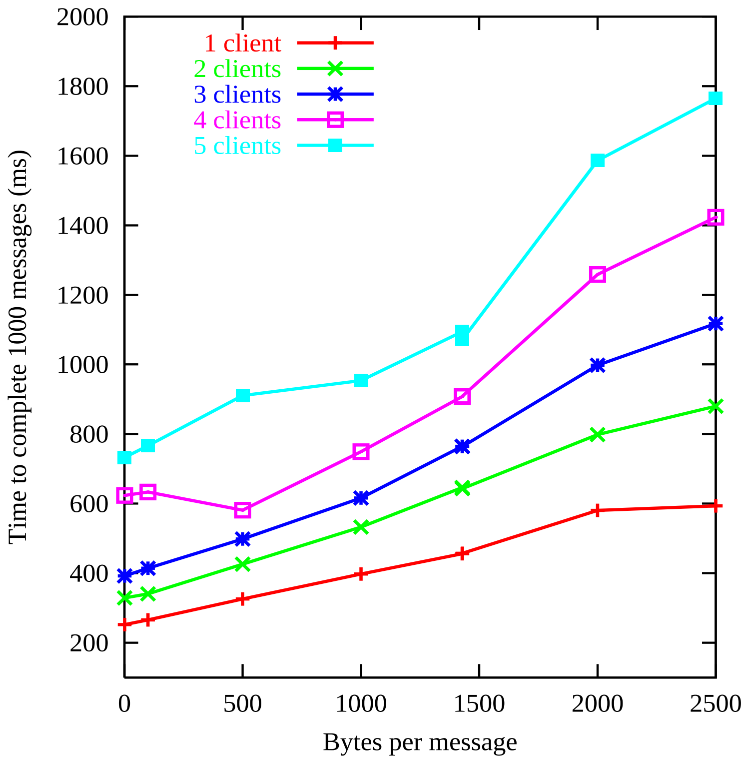

Figure 3.12. Cluster B Growth

This graph is nearly identical to the TCP/IP one except all of the lines are vertically lower, due to the latency decrease.

Figure 3.13 contains the data for Cluster A. Each line behaves more linearly than the other protocols. The non-linear bumps of the 4 and 5 client tests still exist but are much less prevalent. The vertical shifts between lines are less significant and the growth rate is less. Not only are the overall slopes less steep than TCP/IP or Cluster B, but the starting points for each line are significantly lower.

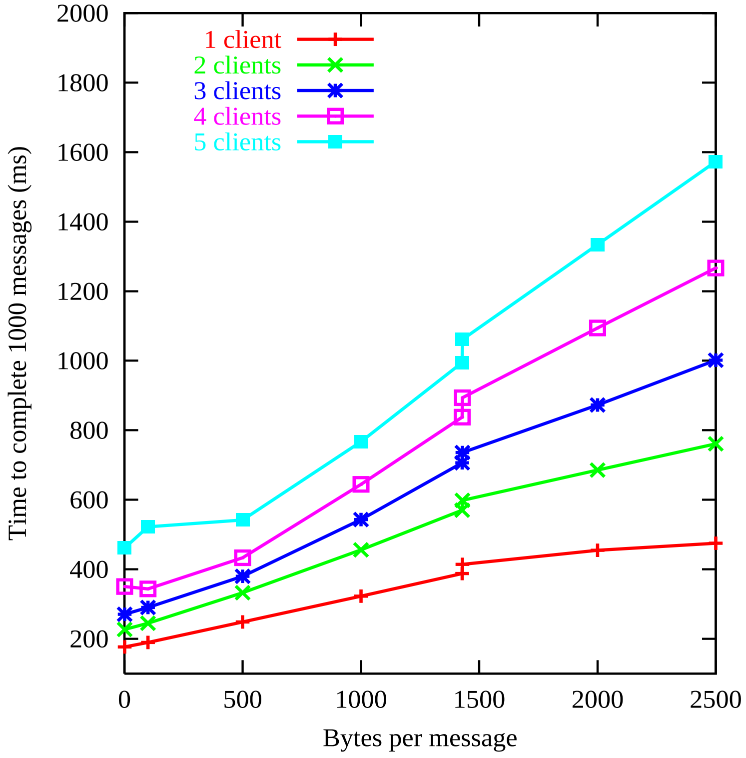

Figure 3.13. Cluster A Growth

3.3.7 Message Growth

Figure 3.14 shows how the time to send a fixed size message changes as the number of clients increases. This is an important graph because it shows how the protocols scale to handle more machines compared to one another. The lines on this graph are not linear because adding another client is more expensive than adding the previous one. This is true from multiple standpoints; adding one more client consumes more network bandwidth, especially if the network is already over-saturated with messages. Also, as the server becomes busier handling requests, the response time to send clients work increases, and that increase is multiplied by the number of clients.

The changes made by Cluster A accounts for a constant performance increase

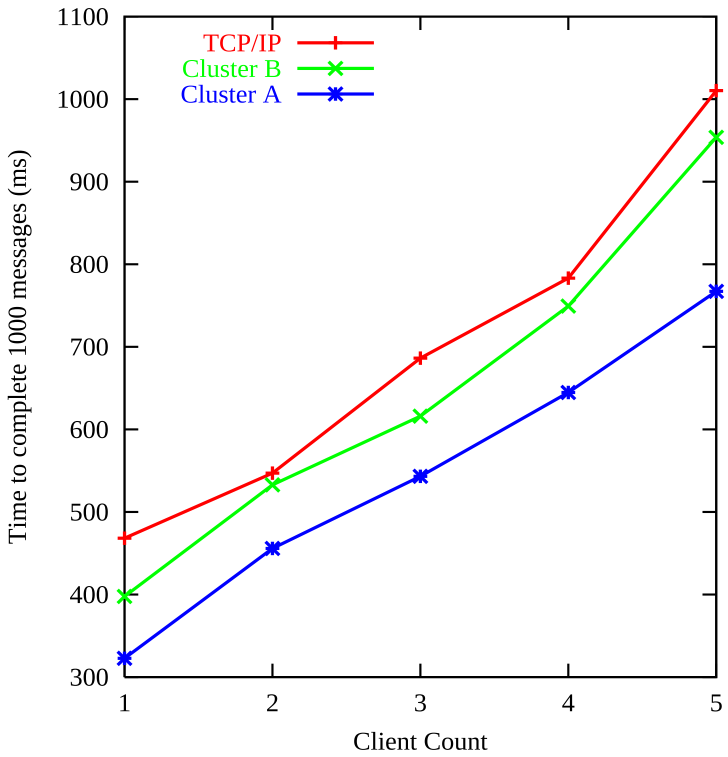

Figure 3.14. 1000 Byte Messages

over TCP/IP as the number of clients grows. The overall growth properties of the protocols are the same; reducing latency helps bring down the overall time but does not affect the asymptotic efficiency of the algorithm.

Chapter 4

CONCLUSION

Neither protocol affects the usage for standard TCP/IP programs. This allows programs to use cluster and TCP/IP protocols simultaneously, satisfying the project's first requirement. Replacing TCP/IP programs with these protocols results in a consistent speed improvement.

The greatest benefit of the Cluster B protocol is a slight performance improvement over TCP/IP without loosing reliability. Though the gains are small, they are practically free, requiring only a minor code change and recompilation.

The Cluster A protocol has an ever greater speed improvement than Cluster B. Figure 4.1 compares the TCP/IP performance for small messages against that of Cluster A, where Cluster A is serving more clients.

Cluster A sends a single byte to three or even four clients faster than the TCP/IP protocol can send that message to two systems. In fact, it can send 500 bytes to four systems faster then TCP/IP can send to two. Also, when sending 1000 bytes, it can send to an additional client for free.

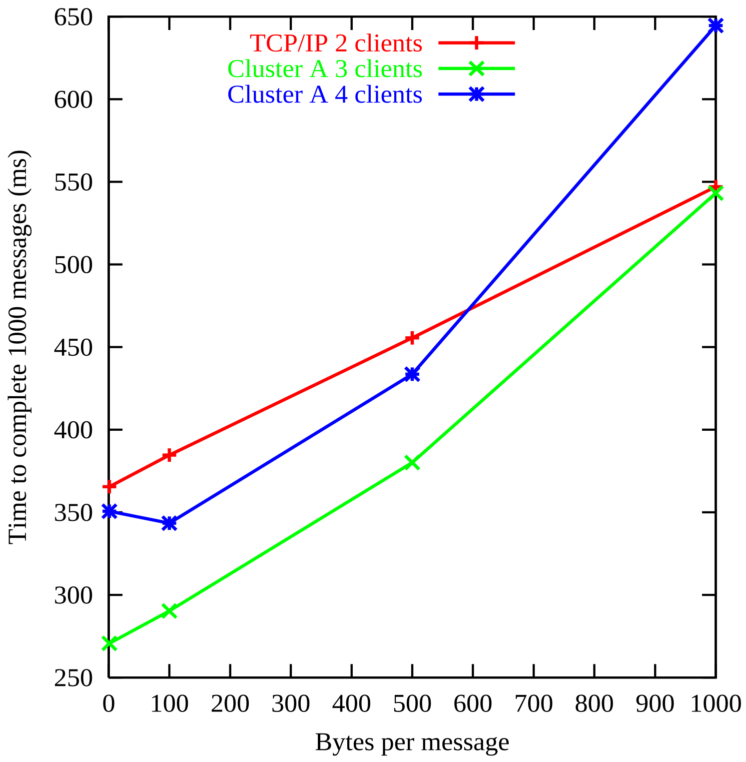

Figure 4.1. Protocol Comparisons

4.1 Limitations

For Cluster B, the only limitation is the protocol must be used on closely networked systems. The biggest limitation for Cluster A is that the network cards drop packets when the network traffic is high. These packet drops occur under TCP/IP, but are not noticed due to the TCP layer requesting resends for out of order data, or ACK timeouts. After modifying the device driver [6] to watch events signaled whenever a packet is recognized, it is evident that the device driver unwittingly misses packets. This means the NIC never notices the packet on the wire and makes no note of it. This violates the initial assumption that the Beowulf's network is reliable. This problem does not affect the Cluster B implementation because it is based on the TCP protocol, but causes problems with the Cluster A when many small packets are transmitted.

The TCP layer has two properties that lessen the likelihood that the NIC will miss packets. The first is the MSS, which drops to zero before the NIC starts missing packets, and the second is the packet merging. If 50 one byte messages were rapidly going to be sent to a client, they would be merged into a single packet, and only 1 message is sent. This differs with Cluster A's approach. Due to the protocol implementation, the 50 packets are sent one after another along the wire as fast as the NIC can transmit. By the fortieth packet or so, the receiving NIC consistently misses a packet, but catches the rest.

This problem prevents Cluster A from being widely adopted, but opens the door for future opportunities to make use of the speed improvement.

4.2 Future Opportunities

After testing the new protocols with the available resources, additional opportunities are apparent for future use and testing.

In order to make widespread use of Cluster A, new hardware should be developed to detect a dropped packet and re-request it from the sender within a certain timeout. This would make the Cluster A protocol fully-reliable.

Another modification to hardware would be supporting packets larger than the current MTU size. This would extend the benefits seen on smaller messages using Cluster A to larger sized messages.

More tests could be done with all the three protocols on a Beowulf cluster with many more systems. It would be interesting to see if the properties scale, or if there are other unexpected issues similar to the network overloading and pinching.

Much of the parallel computing community uses PVM (Parallel Virtual Machine) [8]. It uses UDP packets instead of TCP to send messages between systems. Many of the speed improvements in Cluster A, like the packet ring, can be reused to implement a lower latency UDP message. This would improve the performance of parallel software.

As seen from the system differences between the small and large Beowulf tests, the relational speed of the network plays an important role in the protocol's performance. Porting the cluster code to Gigabit Ethernet would be another opportunity for performance improvement. This should make both cluster protocols perform even better because, as the network time decreases, the latency becomes a larger percentage of the overall time.

Running both Cluster protocols in an SMP (Shared Memory Multi-Processing) environment has not been done. The stock 2.4.18 kernel had network device driver troubles when ran under SMP, so no testing took place.

There are performance benefits available for a Beowulf, which are nearly transparent to the end user. More work needs to be done to utilize the true potential of a low latency system, but the performance benefits of Cluster A should inspire future work.

Chapter 5

REFERENCES

[1] P. Buonadonna, A. Geweke, D. E. Culler. “An Implementation and Analysis of the Virtual Interface Architecture”. *Proc. of Supercomputing ’98.* Orlando, Florida, Nov 1998.

[2] T. von Eicken, A. Basu, V. Buch, W. Vogels. “U-Net: A User-Level Network Interface for Parallel and Distributed Computing”. *Proc. of the 15th ACM Symposium on Operating System Principles.* Copper Mountain, Colorado, Dec 1995.

[3] K. Skevik, T. Plagemann, V. Goebel. “Evaluation of a Zero-Copy Protocol Implementation”. *Proc. of the 27th Euromicro conference - Multimedia and Telecommunications Track.* Warsaw, Poland, Sept 2001.

[4] J. Loncaric. *Linux TCP patches to improve acknowledgment policy.* Aug 2001. `http://research.nianet.org/~Josip/LinuxTCP-patches.html`

[5] A. TanenBaum. *Computer Networks.* 3rd Edition. Prentice Hall PTR, Upper Saddle River, NJ, 1996.

[6] A. Rubini. *Linux Device Drivers.* O’Reilly, Sebastopol, CA, 1998.

[7] W. Stevens. *UNIX Network Programming, Volume 1.* 2nd Edition. Prentice Hall PTR, 1998.

[8] *PVM: Parallel Virtual Machine.* main website: `http://www.csm.ornal.gov/pvm`

[9] T. Sterling, J. Salmon, D. Becker, D.F. Savarese. *How to Build a Beowulf.* MIT Press, 1999.

[10] *Beowulf History.* website `http://www.beowulf.org/beowulf/history.html`

Appendix A

SMALL CLUSTER TESTS

The Beowulf consists of two computers connected via 100 MBit Ethernet. The first system is a P3 400 with 64 megs of RAM and a Kingston KNE110TX 100 MBit NIC. The second is a AMD K3 300 with 64 megs of RAM and a Kingston KNE110TX 100 MBit NIC. All times are in microseconds.

A.1 TCP Comparison

Table A.1 TCP/IP Latency 1 byte 10 times

Stock	Kernel B	Kernel A
99	97	97
96	95	95
95	93	98
96	93	97
96	93	96
96	93	96
96	94	96
96	94	96
96	94	96

Table A.2 TCP/IP Latency 100 bytes 100 times

Stock	Kernel B	Kernel A
632	527	558
654	555	551
625	494	551
569	490	580
650	537	541
650	534	550
647	553	571
569	479	567
625	521	559

Table A.3 TCP/IP Latency 1000 bytes 1000 times

Stock	Kernel B	Kernel A
83199	83392	83690
84631	83457	85483
85612	85668	85959
85267	85296	85226
84344	82860	85299
85548	84844	82644
85904	83596	82652
82568	83068	85322
84634	84022	84534

Table A.4 TCP/IP Latency 50000 bytes 1000 times

Stock	Kernel B	Kernel A
4598301	4598647	4600439
4598930	4600079	4600430
4600427	4599872	4601007
4597907	4598963	4600965
4598980	4599974	4600802
4600661	4599484	4601415
4600828	4601250	4601418
4601024	4600665	4603560
4599632	4599867	4601255

Table A.5 TCP/IP Throughput 1 byte 1000 times

Stock	Kernel B	Kernel A
127087	125277	129711
127931	125169	129532
127177	125156	129827
127840	125643	128939
128418	125036	128852
127150	125349	129155
127885	124943	129316
127099	125328	127358
127573	125238	129086

Table A.6 TCP/IP Throughput 100 bytes 1000 times

Stock	Kernel B	Kernel A
147149	146115	148988
147755	145922	147773
147206	146351	147780
147802	145469	148123
147615	146096	147800
147420	145638	147804
147564	146450	149643
147540	145711	149299
147506	145969	148401

Table A.7 TCP/IP Throughput 1500 bytes 1000 times

Stock	Kernel B	Kernel A
329985	328301	330636
330903	327405	333070
330836	328505	330967
330345	328048	332093
329694	327203	330441
330469	327995	330794
330561	328016	330568
330763	326804	330495
330445	327785	331133

Table A.8 TCP/IP Throughput 10000 bytes 1000 timer

Stock	Kernel B	Kernel A
1091488	1082457	1081549
1086858	1081522	1080777
1086548	1082623	1080676
1087866	1083138	1080476
1087761	1081177	1080653
1087198	1081792	1080712
1086522	1082795	1081106
1087992	1082065	1080899
1087779	1082196	1080856

A.2 Protocol Comparisons

All speed tests were done using 25 messages per test. If any more were used, the NICs had a tendency to drop a packet under the Cluster A protocol. In all cases, the Cluster A protocol was ran with a ring size of 100.

Table A.9 Latency 1 byte 25 times

TCP/IP	Cluster B	Cluster A
127	119	212
126	119	205
127	120	216
127	118	210
126	120	223
127	120	205
127	120	215
131	120	214
127	120	213

Table A.10 Latency 100 bytes 25 times

TCP/IP	Cluster B	Cluster A
214	145	229
215	146	212
162	145	227
215	147	201
214	148	204
161	146	191
159	147	196
162	148	193
188	147	207

Table A.11 Latency 500 bytes 25 times

TCP/IP	Cluster B	Cluster A
334	283	288
326	199	273
346	204	264
323	209	280
325	200	294
335	200	285
322	201	266
320	206	268
329	213	277

Table A.12 Latency 1000 bytes 25 times

TCP/IP	Cluster B	Cluster A
971	840	353
913	831	350
926	854	346
903	842	366
925	838	351
894	827	359
904	827	361
901	836	390
917	837	360

Table A.13 Latency 1500 bytes 25 times

TCP/IP	Cluster B	Cluster A
1706	1568	616
1683	1390	620
1716	1557	603
1608	1543	624
1609	1525	619
1728	1557	603
1737	1555	606
1671	1489	626
1682	1523	615

Table A.14 Latency 3000 bytes 25 times

TCP/IP	Cluster B	Cluster A
3047	2907	1131
3045	2838	1144
2964	2945	1134
2971	2848	1130
2880	2985	1122
2886	2991	1135
2887	2983	1131
3036	2928	1116
2965	2928	1130

Table A.15 Latency 5000 bytes 25 times

TCP/IP	Cluster B	Cluster A
5229	5378	1707
5309	5337	1676
5567	5331	1729
5365	5306	1709
5195	5332	1702
5516	5315	1677
5595	5308	1668
5317	5324	1664
5387	5329	1692

Table A.16 Throughput 1 byte 1000 times

TCP/IP	Cluster B	Cluster A
128029	116870	94166
126955	116627	93928
128015	116842	94016
127025	116101	93941
128079	117067	94135
127468	116076	94158
127130	116383	94162
127807	117002	93977
127564	116621	94060

Table A.17 Throughput 10 bytes 1000 times

TCP/IP	Cluster B	Cluster A
129423	118432	94547
128344	118012	94532
128635	118252	94651
129151	117590	94565
129180	118472	94466
128430	117776	94557
129267	118493	94485
128421	117718	94500
128856	118093	94538

Table A.18 Throughput 100 bytes 1000 times

TCP/IP	Cluster B	Cluster A
147771	136160	112783
147109	136435	112707
147699	136043	112728
147678	136150	112841
146968	135692	112760
147579	136202	112730
147053	136016	112733
147022	136068	112729
147360	136096	112751

Table A.19 Throughput 500 bytes 1000 times

TCP/IP	Cluster B	Cluster A
212208	200311	177849
212757	200827	178202
212024	200283	177959
212335	200931	177860
212726	200778	178040
212811	201219	178122
212266	200580	178045
212009	201296	178041
212393	200078	178015

Table A.20 Throughput 1000 bytes 1000 times

TCP/IP	Cluster B	Cluster A
260084	245556	224582
258758	246679	225778
259074	245613	226250
259301	246019	225816
258917	245628	226079
259970	246374	225755
258876	245455	226156
258573	246063	226376
259194	245923	225849

Table A.21 Throughput 1500 bytes 1000 times

TCP/IP	Cluster B	Cluster A
330959	317027	280599
329747	318025	281052
332223	317212	280565
330940	317511	280952
331084	317026	280533
330471	317244	281043
331086	317955	281001
330189	317273	280530
330837	317409	280784

Table A.22 Throughput 3000 bytes 1000 times

TCP/IP	Cluster B	Cluster A
465735	445342	412006
463567	445394	412235
464174	445695	412079
463193	445311	412438
464229	445251	412022
465066	445465	412137
464584	445564	412079
463591	445776	412247
464267	445475	412155

Table A.23 Throughput 5000 bytes 1000 times

TCP/IP	Cluster B	Cluster A
638612	620522	595339
637937	620323	595906
637956	621303	595382
637931	620424	595842
637982	620953	595302
638474	621639	595987
638647	620721	595395
637998	621012	595709
638192	620862	595608

Table A.24 Throughput 10000 bytes 1000 times

TCP/IP	Cluster B	Cluster A
1086536	1065564	1062368
1086348	1065174	1063266
1086362	1065696	1061082
1086922	1064912	1061779
1087107	1064136	1061251
1087245	1065267	1061366
1087036	1065000	1061920
1086631	1064517	1061644
1086773	1065033	1061835

Appendix B

LARGE CLUSTER

This is a Beowulf of six systems, all using a dual Pentium 166 MHz with 64 Megs of RAM, with a 3Com 3c905 network card, on a 100 MBit network. The SMP mode was turned off because the network drivers were causing troubles on the 2.4.18 kernel. All times are in microseconds.

B.1 TCP/IP

Table B.1 TCP/IP 1 byte 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
286580	373193	440190	700091	770557
293966	357559	443901	700632	758970
293991	368993	440226	695935	769699
290218	361984	445236	686712	762309
294342	358945	433321	691751	761591
288203	370570	446311	689950	771071
292255	365575	435037	700833	766478
291924	367322	441383	686475	770872
291435	365518	440701	694047	766443

Table B.2 TCP/IP 100 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
310371	383570	462036	721113	813414
304417	386365	467906	721582	809126
313910	380279	469856	720262	816332
311840	387551	464267	713086	808002
309719	380405	468940	711558	802625
304685	389675	466719	713796	802766
315029	380556	461777	716405	810830
307152	388103	467535	709459	809123
309640	384563	466130	715908	809027

Table B.3 TCP/IP 500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
381218	453333	553519	654612	979797
374251	450814	555857	642832	986734
382626	451754	555188	668236	981348
377137	467805	559766	651990	983940
382729	452492	550261	669841	977051
373532	459749	559306	655261	973906
376976	450529	551016	669220	974232
378934	458373	554537	662397	976711
378425	455606	554931	659299	979215

Table B.4 TCP/IP 1000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
470858	550158	688894	771957	1004669
465284	541898	682383	779200	1015664
472870	551774	687428	810999	1001842
463771	544121	685405	774066	1013376
465834	545026	682120	793138	1010194
475321	551412	686817	782842	1014607
464106	540697	684927	771134	1004670
468579	550885	691525	782481	1017677
468328	546996	686187	783227	1010337

Table B.5 TCP/IP 1428 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
483562	667387	797461	952363	1100814
483949	666425	791968	938986	1099466
483131	668481	790616	945037	1093610
482638	668590	792861	944792	1100718
482398	666970	789439	942087	1095671
482992	666736	788738	938673	1094992
484690	665367	786916	942675	1100748
482395	664017	787392	939127	1094806
483219	666747	790674	942968	1097603

Table B.6 TCP/IP 1429 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
479324	666783	792714	944828	1102035
485873	665648	792298	948285	1101691
482328	667128	790841	938376	1098513
483847	665878	792710	949322	1099177
485811	668801	792535	946348	1098828
484170	666699	788419	945056	1098190
480235	666526	794626	939426	1097294
479784	666967	794760	943530	1101143
482672	666804	792363	944396	1099609

Table B.7 TCP/IP 2000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
583936	817821	1020583	1348704	1659299
583789	818208	1032896	1312763	1652374
580825	820029	1022883	1352992	1661971
583188	817527	1016238	1345054	1653782
578204	821984	1025065	1321811	1652259
581853	819257	1019739	1321267	1680581
582128	822276	1032168	1342148	1652333
581803	820347	1017953	1363135	1655260
581966	819681	1023441	1338484	1658482

Table B.8 TCP/IP 2500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
695432	903817	1376818	1466796	1848715
687651	905439	1175367	1482461	1795100
698800	902592	1185279	1457866	1850464
685519	901980	1181724	1729840	1833544
688851	902563	1160849	1470261	2002449
688273	905906	1157855	1478249	1808562
690755	901144	1158109	1472027	1806216
681795	905891	1178480	1466177	1775909
689635	903667	1196810	1502960	1840120

B.2 Cluster B

Table B.9 Cluster B 1 byte 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
250137	326233	394187	628284	737987
251337	327782	387190	621328	728819
252857	326014	391334	617600	742059
251314	328994	387447	624179	737324
249720	332145	393267	624945	735717
256775	327036	396218	621627	718559
253303	329764	391938	625407	742078
253954	331037	393767	626589	720255
252425	328626	391919	623745	732850

Table B.10 Cluster B 100 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
264653	339117	414875	649345	764626
264839	338588	413919	607869	763889
265162	341230	416412	652475	770343
263798	338461	411056	644176	764911
268009	341786	413379	650405	773027
265273	340524	416735	597341	766483
263777	342063	412144	644214	767520
268492	341585	414672	622432	760049
265500	340419	414149	633532	766356

Table B.11 Cluster B 500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
325892	426439	497502	582278	927159
325525	428381	497500	580306	885705
324774	422480	500696	581692	921137
328403	426694	495932	581068	891261
326379	424013	498959	579316	925871
326787	427689	497112	582042	886687
322761	425500	502620	583004	924776
324505	425099	496447	578306	921144
325628	425787	498346	581002	910468

Table B.12 Cluster B 1000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
399692	528370	616495	749958	934585
397828	528807	613168	746980	963304
398843	534892	613702	749493	942754
395657	535598	620416	748869	946773
397839	529787	617260	749908	959688
396162	533789	613360	749494	946503
398675	535084	621688	748575	966578
396462	535676	612514	750661	970344
397645	532750	616075	749242	953816

Table B.13 Cluster B 1428 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
456912	646558	766822	909416	1063312
453533	645834	764353	910075	1070195
455583	645857	769561	905655	1070306
455784	646649	761815	909104	1075677
456569	644502	765159	902658	1060173
455379	647693	763359	909041	1064064
454944	646213	762864	910981	1058127
458349	646157	761069	905001	1289253
455882	646183	764375	907741	1093888

Table B.14 Cluster B 1429 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
455210	643022	765657	909510	1068421
457123	643051	762972	911568	1070051
457894	642430	761944	906097	1073806
455923	643671	762889	909545	1064329
458040	641945	760052	908907	1071669
458723	644545	768363	909488	1075021
456973	640621	761721	908415	1079818
455135	643584	767630	908385	1066339
456878	642859	763904	908989	1071182

Table B.15 Cluster B 2000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
575144	803208	997272	1266275	1619086
586112	794304	996824	1266012	1576006
580312	796622	993447	1259225	1572408
585129	799564	1000277	1254060	1583857
580364	800100	994329	1247642	1585986
576625	793397	1000405	1256656	1590705
579365	799755	1001047	1262613	1578161
582136	799444	998651	1254314	1584746
580648	798299	997782	1258350	1586369

Table B.16 Cluster B 2500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
591158	880652	1116117	1418784	1798757
591072	877721	1114455	1409888	1776918
592702	878453	1123108	1410790	1749382
590467	879418	1118265	1459553	1760071
592394	883101	1118253	1410721	1753621
593431	877747	1117870	1432274	1762622
594897	882001	1119033	1410044	1770093
598787	879158	1112535	1431990	1746699
593114	879781	1117456	1423006	1764770

B.3 Cluster A

Table B.17 Cluster A 1 byte 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
174780	227629	270409	353809	469290
177555	226352	272051	355153	459477
177513	227067	268649	350606	459199
180514	228172	271259	353841	465015
175132	225733	271331	347819	457466
174999	225853	270998	343149	459914
174834	228213	269437	351107	458460
178016	227088	270898	349308	461274
176668	227013	270629	350599	461262

Table B.18 Cluster A 100 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
189406	243806	287870	341725	521745
189614	244543	293303	347613	521708
189927	246620	291483	345093	524156
189089	244089	289314	345012	523400
189687	245264	288387	343401	523722
189268	243609	290198	341010	519309
189911	245386	290038	340999	522807
189717	242986	291998	341963	521066
189577	244538	290324	343352	522239

Table B.19 Cluster A 500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
250162	333715	378639	433820	540708
247245	332295	380090	433861	541985
249501	332553	379755	432105	540535
248129	332951	379670	433443	543090
250592	333405	381702	432840	542987
247479	332097	379296	433841	543508
249028	333417	379076	433369	539859
247855	332505	382483	435049	542894
248749	332867	380089	433541	541946

Table B.20 Cluster A 1000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
322538	456304	543786	644864	762041
321894	455015	544479	644479	771262
323583	455155	542989	645101	766583
322247	457207	540797	644304	768302
323012	456755	544667	644871	767898
322962	454450	542668	643837	766394
322068	456717	542125	644150	769430
322605	455490	543399	645233	764581
322614	455887	543114	644605	767061

Table B.21 Cluster A 1428 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
389804	567877	705657	838620	994327
389499	569777	706909	837319	993397
389954	571790	707475	838611	989789
388299	570724	705739	837334	994465
387043	570776	705454	838575	998363
386154	571222	705006	837942	990164
385049	568275	707291	837223	997818
387345	569392	705982	837506	998876
387893	569979	706189	837891	994650

Table B.22 Cluster A 1429 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
414897	598716	735034	896172	1056891
413411	597024	735171	893582	1070781
415946	597589	736888	889354	1063363
413761	600599	734508	896817	1054319
415326	596770	737088	901336	1051379
412798	598020	737732	888997	1059668
414411	599406	733108	890149	1063519
413539	599349	736823	889267	1070075
414261	598434	735794	893209	1061249

Table B.23 Cluster A 2000 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
454679	685137	873147	1091502	1342201
454537	685389	872465	1091109	1344213
456484	686198	872945	1092779	1338179
454549	686859	873151	1090220	1334532
453891	683615	873443	1096570	1333460
454428	684393	871826	1094920	1327914
453559	684924	873826	1102039	1331288
455390	686776	872318	1094037	1322925
454690	685411	872890	1094147	1334339

Table B.24 Cluster A 2500 bytes 1000 times

1 client	2 clients	3 clients	4 clients	5 clients
473154	759220	996103	1265717	1572605
474749	759294	1004925	1260263	1581993
473739	762277	1008458	1263747	1565424
477217	758527	1001107	1265670	1572338
475399	764796	999628	1268055	1567391
472569	761362	994321	1267125	1570988
475499	759284	1001308	1273936	1576842
476241	761566	1008078	1270167	1574933
474821	760791	1001741	1266835	1572814

Appendix C

SKB STRUCTURE

The only way to track data within the protocol is to use the SKB structure. If this structure is modified, then all of the network device drivers would need to be modified. Instead, a section of the SKB structure is reused to implement the Cluster A protocol. The `cb` data is a reusable data structure in the SKB, and the cluster A protocol uses it to store the length of the transmission ring and the number of times this packet was waiting to be reused (Implying the transmission ring was overrun). This code is located in `linux/include/linux/skbuff.h`.

```
struct sk_buff {
  /* Next buffer in list */
  struct sk_buff  * next;

  /* Previous buffer in list */
  struct sk_buff  * prev;

  /* List we are on */
  struct sk_buff_head * list;

  /* Socket we are owned by */
  struct sock     *sk;

  /* Time we arrived */
  struct timeval  stamp;

  /* Device we arrived on/are leaving by */
```

```
struct net_device       *dev;

/* Transport layer header */
union
{
 struct tcphdr   *th;
 struct udphdr   *uh;
 struct icmphdr  *icmph;
 struct igmphdr  *igmph;
 struct iphdr    *ipiph;
 struct spxhdr   *spxh;
 unsigned char   *raw;
} h;

/* Network layer header */
union
{
 struct iphdr    *iph;
 struct ipv6hdr  *ipv6h;
 struct arphdr   *arph;
 struct ipxhdr   *ipxh;
 unsigned char   *raw;
} nh;

/* Link layer header */
union
{
 struct ethhdr   *ethernet;
 unsigned char   *raw;
} mac;

struct  dst_entry *dst;
/*
 * This is the control buffer. It is free to use for every
 * layer. Please put your private variables there. If you
 * want to keep them across layers you have to do a skb_clone()
 * first. This is owned by whoever has the skb queued
 * ATM. */
 char            cb[48];

/* Length of actual data */
unsigned int    len;
unsigned int    data_len;

/* Checksum */
```

```
unsigned int    csum;

/* Dead field, may be reused */
unsigned char   __unused,

   /* head may be cloned (check refcnt to be sure). */
   cloned,

   /* Packet class */
   pkt_type,

   /* Driver fed us an IP checksum */
   ip_summed;

   /* Packet queueing priority */
   __u32   priority;

   /* User count - see datagram.c,tcp.c */
   atomic_t        users;

/* Packet protocol from driver.*/
unsigned short  protocol;

/* Security level of packet */
unsigned short  security;

/* Buffer size */
unsigned int    truesize;

/* Head of buffer */
unsigned char   *head;

/* Data head pointer */
unsigned char   *data;

/* Tail pointer */
unsigned char   *tail;

/* End pointer */
unsigned char   *end;

/* Destruct function */
void            (*destructor)(struct sk_buff *);

#ifdef CONFIG_NETFILTER
```

```
    /* Can be used for communication between hooks. */
    unsigned long   nfmark;

    /* Cache info */
    __u32           nfcache;

    /* Associated connection, if any */
    struct nf_ct_info *nfct;

    #ifdef CONFIG_NETFILTER_DEBUG
       unsigned int nf_debug;
    #endif
  #endif /*CONFIG_NETFILTER*/

  #if defined(CONFIG_HIPPI)

  union{
    __u32   ifield;
  } private;

  #endif

  #ifdef CONFIG_NET_SCHED
       __u32           tc_index;  /* traffic control index */
     #endif
};
```