User-Level TCP in High-Performance Server-Cluster Networks

A dissertation submitted for the degree of Doctor of Philosophy

Kieran Mansley, Fitzwilliam College

December 2004

LABORATORY FOR COMMUNICATION ENGINEERING
Department of Engineering
University of Cambridge
Abstract

Over the past 20 years, the Internet has grown from a small research network to a network of global importance. As Internet traffic continues to grow, the servers that provide the content must scale to be able to deliver the next branch of services, while remaining compatible with the millions of clients that already make use of them. Of central importance to this is the TCP/IP suite of protocols that allow hosts across the Internet to communicate.

This dissertation argues that recent developments in user-accessible network interfaces and the use of clusters of servers (for both reliability and performance) allow for a new architecture of network protocols. Many of the problems that limit the performance of server network access are due to the way the protocols and operating systems are structured.

To this end, an analysis of the overheads in traditional network architectures and a new structure to take full advantage of a user-accessible network are presented. The interface to the network itself, and how to transparently integrate a network stack into an application are given particular attention. To demonstrate this, the Cambridge User Level TCP (CULT) has been created and shown to outperform traditional networks.

Clustering of servers allows further scope for the involvement of the network itself in reducing the overhead of network access. This dissertation demonstrates how common network components, such as a gateway, duplicate specific functions of higher layer protocols, and how (with suitable integration) this functionality can be realised more efficiently. This is demonstrated through the use of the
cluster gateway to buffer and retransmit TCP segments at little additional cost to itself, and considerable saving to the end nodes.

The combination of TCP at user level and a low-latency network provides a good platform for load balancing. The gateway developed to bridge CULT to the Internet can also be used to load balance requests across the cluster. This dissertation presents a low-overhead and high-accuracy technique for measuring the load of a cluster node that is using the CULT stack and compares this to standard methods.

In summary, this dissertation examines a full vertical slice of the system from application to network gateway in a server cluster. It describes how the cluster network and gateway can be used to make Internet access easier and more efficient for the nodes.
Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty five thousand words, including tables, footnotes, and bibliography.
Publications

Some of the work described in this dissertation has been published elsewhere:


Trademarks

All trademarks are the property of their respective owners.
Acknowledgements

I am indebted to my sponsors who generously funded this work: AT&T Laboratories — Cambridge Ltd, and the Royal Commission for the Exhibition of 1851.

A great many individuals have proved invaluable through the course of this project. This work would not have been possible without the hardware and software provided by the CLAN project at AT&T Laboratories — Cambridge Ltd, and from Level 5 Networks Ltd. In particular Steve Pope, Derek Roberts and David Riddoch have gone out of their way to help whenever possible, and continued to do so when their own involvement in the project ended with the closure of AT&T Laboratories in Cambridge.

Andy Hopper, as my supervisor, has provided excellent advice and encouragement whenever required. He should also be credited for creating the opportunity for me to participate in this research, both at AT&T Laboratories and the Laboratory for Communication Engineering.

David Belanger, at AT&T Research in Florham Park, happily stepped in as my Industrial Supervisor to rescue this project when AT&T Laboratories in Cambridge closed. He then ensured that AT&T continued to provide the necessary support and funding to allow me to complete and publicise this research.

Finally, many members of the Laboratory of Communication Engineering and beyond have ensured that I have maintained a healthy perspective throughout the last few years, as well as providing excellent feedback on research ideas and proofreading services.
## Contents

1 Introduction .................................................. 1
   1.1 The Problem .............................................. 1
   1.2 Contribution ............................................. 3
       1.2.1 Scope ............................................... 4
       1.2.2 Extent of Collaboration ............................ 4
   1.3 Outline ................................................... 5

2 High-Performance Server-Cluster Networks ..................... 6
   2.1 Metrics for High-Performance Networks .................. 6
   2.2 Server Clusters ........................................... 10
   2.3 Sources of Network Overhead ............................. 12
       2.3.1 Network Interface .................................. 12
       2.3.2 Protocols ............................................ 14
       2.3.3 Sockets API and the Application/OS Interface .... 18
   2.4 Dealing with Overload ..................................... 20
       2.4.1 Receive Livelock ..................................... 20
       2.4.2 Congestion Control and Checksums ................. 22
       2.4.3 Load Balancing ...................................... 25
   2.5 Networks for Server Clusters .............................. 27
       2.5.1 High-performance networking concepts .............. 28
       2.5.2 Gigabit Ethernet .................................... 28
       2.5.3 Myrinet .............................................. 29
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1 Update the Sockets API</td>
<td>83</td>
</tr>
<tr>
<td>4.2.2 Alterations to the Transport Protocols</td>
<td>83</td>
</tr>
<tr>
<td>4.2.3 Blocking semantics</td>
<td>84</td>
</tr>
<tr>
<td>4.2.4 Memory Multicast</td>
<td>85</td>
</tr>
<tr>
<td>4.2.5 Copy On Write</td>
<td>86</td>
</tr>
<tr>
<td>4.2.6 Discussion</td>
<td>87</td>
</tr>
<tr>
<td>4.3 Protocol Gateway and Retransmission</td>
<td>87</td>
</tr>
<tr>
<td>4.3.1 Gateway Buffering for Retransmission</td>
<td>88</td>
</tr>
<tr>
<td>4.3.2 Implementation</td>
<td>90</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>93</td>
</tr>
<tr>
<td>4.5 TCP/IP Performance</td>
<td>93</td>
</tr>
<tr>
<td>4.5.1 Test bed</td>
<td>93</td>
</tr>
<tr>
<td>4.5.2 NetPIPE tests</td>
<td>94</td>
</tr>
<tr>
<td>4.5.3 Latency</td>
<td>98</td>
</tr>
<tr>
<td>4.5.4 TCP Retransmission by the Gateway</td>
<td>99</td>
</tr>
<tr>
<td>5 Load Balancing</td>
<td>103</td>
</tr>
<tr>
<td>5.1 Existing Approaches and Features</td>
<td>103</td>
</tr>
<tr>
<td>5.1.1 Obtaining Load Measurements</td>
<td>105</td>
</tr>
<tr>
<td>5.2 Proposed Alternative Approach</td>
<td>106</td>
</tr>
<tr>
<td>5.3 Implementation</td>
<td>108</td>
</tr>
<tr>
<td>5.3.1 Node Assignment</td>
<td>109</td>
</tr>
<tr>
<td>5.3.2 Support for Standard Ethernet Networks</td>
<td>110</td>
</tr>
<tr>
<td>5.4 Load Balancing Performance</td>
<td>112</td>
</tr>
<tr>
<td>5.4.1 Test bed</td>
<td>112</td>
</tr>
<tr>
<td>5.4.2 Node Assignment</td>
<td>113</td>
</tr>
<tr>
<td>5.4.3 Httperf</td>
<td>116</td>
</tr>
<tr>
<td>5.4.4 Bandwidth Overhead</td>
<td>123</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>123</td>
</tr>
<tr>
<td>6 Timers</td>
<td>124</td>
</tr>
<tr>
<td>6.1 Timer Implementation Problems</td>
<td>124</td>
</tr>
<tr>
<td>6.1.1 Historical Constraints</td>
<td>125</td>
</tr>
</tbody>
</table>
CONTENTS

6.1.2 Timer Implementations .................................... 126
6.1.3 Time at User Level ........................................... 127
6.2 Cross-Thread Scheduling ....................................... 128
6.3 CULT Timer Architecture ....................................... 130
   6.3.1 Fast-Timer Buckets ....................................... 130
   6.3.2 Lazy Implementation ....................................... 131
   6.3.3 Implications ............................................... 133
   6.3.4 Timer Threads .............................................. 134
   6.3.5 CULT Timers Summary ..................................... 135
6.4 Measurements and Results ..................................... 135
   6.4.1 Cost of Queueing an Acknowledgement .................. 136
   6.4.2 Cost of Fast Timer Routine .............................. 137
   6.4.3 Reduction in Data Path Locks ........................... 138
   6.4.4 Discussion and Comparison .............................. 138
6.5 Summary ...................................................... 141

7 Conclusions ......................................................... 142
   7.1 Contributions ............................................... 142
   7.2 Further Work ................................................. 143
   7.3 Outlook ....................................................... 145
List of Figures

2.1 A Server Cluster ............................................. 10
2.2 Store-and-Forward vs Cut-Through ............................ 30
2.3 Distributed Message Queue ................................. 37

3.1 Queues on Thread Boundaries ................................. 53
3.2 Structure of a traditional network stack ..................... 54
3.3 Structure of a typical user-level network stack ............... 55
3.4 Structure of the Arsenic stack ................................. 56
3.5 Opening a TCP Connection ..................................... 57
3.6 Closing a TCP Connection ..................................... 57
3.7 Structure of the CULT stack ................................. 59
3.8 Flow of Execution through CULT code ....................... 61
3.9 Received Data Queues ......................................... 63
3.10 TCP handshake processing. ................................... 68

4.1 A Virtual Ring Buffer ......................................... 80
4.2 Moving buffering for retransmission from the stack to the gateway 89
4.3 TCP retransmission option header ............................ 91
4.4 The test networks used to compare the CULT stack and gateway with a standard Gigabit Ethernet equivalent. ................. 95
4.5 NetPIPE Performance of CULT and Gigabit Ethernet .......... 96
4.6 Costs of copy and free operations ............................ 101
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Ethernet Connection Protocol Header</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>The test bed network</td>
<td>113</td>
</tr>
<tr>
<td>5.3</td>
<td>Distribution of TCP lifetimes for the different load-balancing metrics</td>
<td>118</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of round-robin and buffer-usage metrics</td>
<td>121</td>
</tr>
<tr>
<td>6.1</td>
<td>Scheduling States of a Process</td>
<td>128</td>
</tr>
<tr>
<td>6.2</td>
<td>Timer Buckets</td>
<td>131</td>
</tr>
<tr>
<td>6.3</td>
<td>Time Taken to Queue a Delayed ACK</td>
<td>136</td>
</tr>
<tr>
<td>6.4</td>
<td>Overhead of iterating over a list of connections</td>
<td>139</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Impact of read frequency and size on performance . . . . . . . . . 65
4.1 NetPIPE latencies . . . . . . . . . . . . . . . . . . . . . . . . . 98
4.2 Ping latencies . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
4.3 Costs of copying payloads . . . . . . . . . . . . . . . . . . . . . 100
4.4 Cost to the gateway of examining each packet . . . . . . . . . . . 100
5.1 Ethernet connection protocol header details . . . . . . . . . . . . 111
5.2 Mean time in ns to determine which node to use . . . . . . . . . . 114
5.3 Mean time in ns to sort list after connection removal . . . . . . . 116
5.4 Mean time in ns to update load indicator . . . . . . . . . . . . . . 116
5.5 TCP connection lifetimes for different load-balancing metrics . . 117
5.6 Response times for different load-balancing metrics . . . . . . . 119
5.7 Percentage of requests satisfied for uniform and non-uniform clus-
ters using different load balancing metrics . . . . . . . . . . 120
List of Abbreviations

ACK  A TCP ACKnowledgement.

API  Application Program Interface: a set of functions one module of code can use to access another module of code.

AQM  Active Queue Management.

ARP  Address Resolution Protocol.

ATM  Asynchronous Transfer Mode: a cell switched network.

BGP  Border Gateway Protocol.

CLAN  Collapsed LAN: a user-level network from AT&T Laboratories.


CULT  Cambridge User-Level TCP: a product of this work.

CPU  Central Processing Unit: the main CPU in a computer.

CRC  Cyclic Redundancy Check.

DAT  Direct Access Transport: a new high-performance protocol to replace the sockets API.

DDP  Direct Data Placement: an IETF draft standard for RDMA networking.

DMA  Direct Memory Access.
LIST OF ABBREVIATIONS

DMQ  Distributed Message Queue.

DSM  Distributed Shared Memory.

ECN  Explicit Congestion Notification.

ETA  Embedded Transport Acceleration: an Intel project dedicating a CPU within a SMP server to network access.

FIFO  First In, First Out: a type of queue.

FIN  A TCP control packet sent to close a connection.

FPGA  Field-Programmable Gate Array.

GigE  Gigabit Ethernet.

HTTP  Hyper Text Transfer Protocol.

ICMP  Internet Control Message Protocol: used to support “pings”.

IP  Internet Protocol.

ITR  Interrupt Throttle Rate.

LAN  Local Area Network.

LCE  Laboratory for Communication Engineering (at the University of Cambridge).

MAC  Media Access Control.

MSL  Maximum Segment Lifetime.

MSS  Maximum Segment Size.

MTU  Maximum Transmission Unit: the largest supported segment size.

NIC  Network Interface Card.

OOB  Out Of Band message.
LIST OF ABBREVIATIONS

OS  Operating System.

PCB  Protocol Control Block (not printed circuit board)

PCI  Peripheral Component Interconnect: a bus into which cards such as NICs are connected.

PIO  Programmed Input/Output.

QoS  Quality of Service.

RAM  Random Access Memory.

RDMA  Remote Direct Memory Access.

RED  Random Early Detection.

RFC  Request For Comments: a standardisation document.

RPC  Remote Procedure Call.

RTO  Retransmission Time Out (in TCP).

RTP  Real-time Transport Protocol.

RTT  Round Trip Time.

SAN  System Area Network.

SCI  Scalable Coherent Interface.

SDP  Sockets Direct Protocol: a new high-performance protocol to replace the sockets API.

SMP  Symmetric Multi Processor: a computer with more than one CPU.

SYN  A TCP control packet sent to create a connection.

TCP  Transmission Control Protocol.

TLB  Translation Lookaside Buffer: a list of memory pages and their current locations.
LIST OF ABBREVIATIONS

TOE  TCP Offload Engine.

UDP  User Datagram Protocol.

VIA  Virtual Interface Architecture.

VMMC Virtual Memory Mapped Communication.

VRB  Virtual Ring Buffer.

WAN  Wide Area Network.
CHAPTER 1

Introduction

This chapter provides an introduction to this dissertation, outlining the problems it is trying to solve and what the contributions of this work are.

1.1 The Problem

As Internet traffic increases and network bandwidths rise, servers have to scale to be able to meet demand for content. Transferring data to and from the network and processing the network protocols represents a significant portion of the work expended by servers on the Internet. Some components of this “network overhead” do not decrease in line with Moore’s law led increases in processor speeds, and so they can start to dominate the workload. This can lead to situations such as “receive livelock” where the server spends all its time servicing the network and has no capacity left for performing useful work. The portion of the server that must be used to service the network therefore represents a significant cost.

User-level networking has been suggestion as one way to avoid the restrictions that a traditional kernel places on the implementation of a network stack and the inefficiencies of entering and leaving the kernel on each operation. There are, however, a number of issues that need to be addressed in order to make it viable.
1.1. The Problem

Firstly, there is the issue of interfacing the network stack to the application. While a number of efficient user-level network APIs have been developed, none has yet been widely adopted (although there have been recent attempts to standardise, for example, VIA), largely because of incompatibility with existing applications and the resistance of programmers to use a new model. The sockets API therefore remains the de facto standard, for good reasons, but due to a number of intrinsic limitations it is not well tailored for high performance. A consideration of how its implementation can be improved, without losing compatibility, would be highly beneficial.

Secondly, the structure of the stack itself is a key factor. Much of the existing work on creating protocol stacks at user level has been based on minimal ports of kernel stacks, which are sub-optimal in a user-level environment. Much of the overhead of a protocol such as TCP is not the protocol itself, but the surrounding components that are used to integrate it into the rest of the system.

Thirdly, the way the stack interfaces to the network is also of importance. High performance can be achieved by making the stack tightly bound to exotic hardware, but this can be expensive and does not lend itself to wide scale deployments and interoperability with existing systems. In addition, issues such as reliability and configuration often mean users fall back to using the more familiar kernel TCP over their exotic hardware. A system that is usable across a variety of user-accessible hardware\(^1\) and yet is still able to perform well would clearly be beneficial.

Finally, the impact of these changes on how a group, or cluster, of servers interfaces to the wider Internet has not been widely considered. This problem is two-fold: \(i\) how can a network component reduce the overhead at the cluster nodes without itself becoming a bottleneck; and \(ii\) how can modern network technologies improve the implementation of functionality such as load balancing?

In conclusion, an approach is needed that examines the entire system, from cluster gateway, through hardware APIs, software protocol structure and application level interfaces, and attempts to make best use of each component without tying itself to any one technology.

\(^1\)Abstracting hardware to a common interface is usually the role of the kernel, but it clearly can’t do this if the hardware is accessible at user level.
1.2 Contribution

This work uses hardware developed as part of the CLAN project (at the LCE and former AT&T Laboratories-Cambridge) and hardware from Level 5 Networks Ltd. to explore the implementation of network protocols at user level.

The hardware used is accessible by applications using a novel programming API that bypasses the kernel. This dissertation explores how this API can be best used to support protocols such as TCP.

This dissertation makes the following contributions to the field of high-performance server-cluster networks:

• An analysis of TCP network stack architectures leading to a new “threadless” architecture and prototype implementation for user-level TCPs.

• A novel approach to TCP retransmission that removes the burden of buffering from the stack and utilises functionality of the network gateway to take over this role.

• A discussion of load balancing techniques for a web server cluster and an implementation of a load balancer based on a new load metric: network buffer usage.

• A innovative method for implementing TCP timers without using a separate thread, which removes much of the per-endpoint locking and list searching overhead.

It is the thesis of this dissertation that: (i) TCP at user-level can benefit from a different structure to traditional implementations; (ii) network components in a cluster, such as the gateway, can remove specific burdens from the TCP in the end nodes, without terminating the TCP connection and without holding TCP state for each connection; (iii) within such a cluster, network buffer usage is a good load-balancing metric; and (iv) the combination of these techniques leads to increased TCP performance and server utilisation.
1.2. Contribution

1.2.1 Scope

The entire network system from application to gateway is clearly large and complex. A full detailed analysis of this system would therefore be unrealistic given the time constraints of a Ph.D. and so some aspects receive more attention in this dissertation than others.

The emphasis for this work has been to explore the ways that changes in the implementation of individual protocols and components can yield benefits for the system as a whole. As the main transport protocol used on the Internet, TCP has received the majority of the implementation effort. Other similar (and in many cases simpler) protocols, such as UDP, are also considered, but not to the same extent.

Functionality of the protocols and APIs used (such as the Posix sockets API) has been implemented as and when required. This means that certain (more esoteric) aspects of these will not have been either implemented or widely tested. In particular, some aspects, while necessary to support more novel areas of the research, are in themselves already well researched topics. These areas are highlighted with related work where appropriate but the implementation presented here may not be complete, so as to avoid duplicating others work.

The implementation of network components (such as the gateway) are prototypes to illustrate the principles discussed, rather than production-quality devices.

Finally, as discussed in Section 1.2.2, the network hardware used is supplied by third parties. It is the way this hardware is used that forms the author’s contribution.

1.2.2 Extent of Collaboration

The hardware and associated drivers used by the author were developed in collaboration with the CLAN project (from the former AT&T Laboratories-Cambridge) and Level 5 Networks Ltd, most notably Steve Pope, Derek Roberts and David Riddoch. This includes the “Distributed Message Queue” API which was joint work by the author and the above groups. The work described in Chapters 3, 4, 5 and 6 is substantially the author’s own work.
1.3 Outline

The rest of this dissertation is structured as follows:

Chapter 2 supplies some background to the field of high-performance networks for server clusters, metrics for measuring performance and sources of overhead, along with existing approaches to addressing these issues. It also surveys a range of network technologies aimed at the server cluster, TCP implementations and approaches to implementing timers.

Chapter 3 outlines a number of existing approaches to structuring a network protocol stack and focuses on how the move to user level affects this. It presents the Cambridge User-Level TCP (CULT) and describes its implementation in detail.

Chapter 4 describes the data transmission and delivery operation of CULT and discusses the role network components such as the gateway can play in TCP processing. This includes a novel approach to TCP retransmission that results in benefits throughout the stack and associated interfaces.

Chapter 5 extends the role the gateway plays by turning to load balancing. Here, the novel network technologies employed allow for innovative techniques to be used. These centre on the use of network buffer usage to provide a reliable and low latency indication of load. This is compared to existing methods.

Chapter 6 examines the impact that the stack being at user level has on the implementation of protocol timers. It also describes the limitations imposed by the CULT structure. An implementation of a new timer architecture based on lazy operations is described and its performance is compared to current approaches such as timer wheels.

Performance measurements and discussion are presented at the end of Chapters 4, 5 and 6. Finally, Chapter 7 concludes by summarising the contributions and results from this dissertation and discussing possible avenues for further work.
High-Performance Server-Cluster Networks

This chapter provides some background in the area of high performance networks for server clusters. It describes the weaknesses in traditional approaches that have led to the research presented in this dissertation and provides a survey of existing research and technologies in this field.

2.1 Metrics for High-Performance Networks

Before considering the different network technologies it is important to establish some means to compare them. This section describes a number of properties of networks which can be used to measure their performance.

Bandwidth is the most widely known measure of a network’s performance. It corresponds to the maximum data rate (i.e. number of bits that will travel past a point in the network per second). Bandwidth can be affected by a number of different aspects of the network. There is the hard limit of the line speed of the link. This is typically (for the types of networks being considered) 1 Gbps or more. There are however many sources of network overhead and so this speed is rarely achieved. In particular where per-

1“Network overhead” is the cost of transmitting and receiving data via the network.
packet overheads are high in comparison to per-byte overheads, the achievable bandwidth will be a function of the packet size and will only tend to the maximum as packet sizes become large.

Should more than one connection be using the same network link then typically they will have to share it, again resulting in the application seeing less bandwidth than it might expect. Higher layer protocols, such as TCP, will control the network usage to prevent the network becoming overloaded and attempt to divide it fairly among the users. There has been much work of late to update these congestion control algorithms to ensure they perform well in modern high bandwidth networks [49].

Bandwidth is of particular interest to applications which wish to transfer large amounts of data from one point to another, as in this case it will be the limiting factor on the total time taken.

**Latency** measures the time it takes for a packet to cross a network, from sender to receiver. It includes the overhead of transferring the packet from the application to the network at the sending node (and vice versa at the receiver) as well as the time it spends “in-flight”. For large packets latency can become heavily dependent on bandwidth and so it is most often measured using very small packets. Round trip time (RTT) is the time taken for a packet to traverse a network and a reply to be sent back. This will be approximately twice the latency.

Latency is important to pairs (or more) of applications which are tightly coupled and will typically spend much of the time waiting for a request or response from the other. Clearly if it is possible to reduce the latency of the network the time spent waiting will also be reduced, resulting in faster applications.

In addition, the control loop of many protocols is driven by small packets. For example, the use of acknowledgements in TCP. Any delay that these packets experience could have a disproportionate (and detrimental) effect on the throughput of useful data.

*Host Resources* To make use of a network requires the use of some of the re-
sources in the communicating nodes. There are many resources used, but the most important to consider are the CPU usage, memory usage and resources on the NIC.

CPU cycles are consumed in a number of ways. The processing of network protocols, for example, will involve thousands of instructions. CPU cycles are not limited to the number required to get the work done; many are lost indirectly through inefficiency in the architecture. This will include cache misses, context or thread switches and overhead for servicing interrupts. The CPU cycles used are often separated into a number of different classes. For example, cycles can be labelled as per-packet overhead, per-byte overhead, or the overhead of dealing with each connection. CPU cycles are of particular interest to applications as the CPU must be shared between the process servicing the network (on behalf of the application(s)) and the application(s) that wish(es) to communicate. Each cycle consumed by the network process is effectively taken from the application(s), so reducing the speed at which useful work can be done.

Memory (in most cases RAM) is also required to access the network. This can again be classified into per-connection and per-byte requirements. The amount required will vary depending on the protocol used and the amount available will often be a limit on the achievable bandwidth. Two areas where large amounts of memory are often required are the buffering of packets for retransmission and the buffering of data received from the network while they wait to be processed. Per-connection memory is required to maintain the state for each connection. The number of times the memory is accessed will also have an impact as the speed of the memory bus is often limited.

NICs have, until recently, been based on relatively simple hardware, and so have had very few resources available to them. Some attempts have been made to reduce the impact on the host’s own resources (such as those outlined above) by providing processors and memory on the NIC. This is usually successful in its aim, until the “on-board” resources are all consumed. At this point the lack of resources can represent an artificial limit
on the network, despite the host having spare capacity. For example, if per-connection resources are required on the NIC there will be a limit on the number of concurrent connections. Similarly, as the on-board processor will usually be less powerful than the server’s main processor, the NIC can itself become the bottleneck [85].

These “offload” techniques can be classed as either stateless and stateful: those that require state on the NIC (such as a full TCP Offload Engine (TOE) — discussed further in Section 2.6.6) are stateful, whereas those that have no persistence on the NIC (e.g. checksum offloading) are stateless.

Cost will clearly be affected by many of the factors described above, rather than affecting the performance of the network directly. Very often the choice of network boils down to an economic decision of which offers the best value for money. General purpose networks are very much skewed toward the lower-cost end of the market, but those networks that offer an increase in performance can charge a significant premium to specialist markets. When comparing different network technologies it is an important consideration.

In recent years the impressive increase in the raw speed of networks have resulted in a change of the source of bottlenecks. Applications (and users) are typically concerned with how long it will take to get a certain amount of data from one point in the network to another. This will be determined by a combination of the latency and bandwidth of the network. On low bandwidth links it was dominated by the time to transmit the data on the physical link and as a result increases in bandwidth have resulted in improved performance. However, as bandwidth has increased by orders of magnitude, the sources of overhead that lead to high latency have begun to dominate. This effect is magnified in clusters and LANs where the bandwidth available is typically much greater than that across a WAN or the Internet.

The significance of this trend is that the continued increase in bandwidth from 1Gbps to 10Gbps and beyond will have considerably less impact on the performance seen by applications and users than bandwidth increases have in the past.

---

2Clearly it is possible to engineer hardware with enough resources so that this is not a problem, but this may be prohibitively expensive.
2.2. Server Clusters

The theoretical bandwidth will no longer be realised in practice using the current approach to networking as bandwidth is often no longer the bottleneck.

In order to pass on the benefits of these very high performance networks it is necessary to focus on the new bottlenecks: those caused by the overhead of processing the data at the network nodes.

2.2 Server Clusters

The target of this research is a server cluster. A server cluster is typically a number of powerful computers connected together via a high-performance network (the “internal network”). It will, if dealing with clients in other places, also have a network connection to the Internet (the “external network”). One possible configuration is illustrated in Figure 2.1.

![Figure 2.1: A Server Cluster](image)

Server clusters are put to a number of uses. These can be classified according the resource they (relatively) use the most of:
2.2. Server Clusters

CPU. This type will be limited by the amount of time it takes to do the computations on each node. There is a limited amount of communication between each node and between the cluster and the Internet. For example, when rendering frames for an animated film such as Toy Story each node can be given a separate frame to work on. They do not need to exchange information in order to perform the rendering, and so the network between the nodes is not heavily used. The amount of computation required is large and so this dominates.

Internal Network. This type will be limited by the network between the nodes in the cluster. This could be either due to a lack of bandwidth, or poor latency. The need for communication between the nodes is typically a property of the problem the cluster is being used to solve. Many scientific simulations and computation fall into this category. For example, solving the n-body gravitational problem [98], where each node is responsible for one of the bodies requires the nodes to communicate the location of its mass to all the others at the end of each iteration. The use of the internal network is therefore high compared to the CPU resources used and the external network use (to submit the job and return the results) is minimal.

External Network. If the cluster is communicating with clients across the Internet, the network link between the cluster and the client can often be the limiting factor. This will arise when a large number of requests are made, each request does not require a large amount of CPU to compute and each request can be dealt with by a single node with limited communication within the cluster. An example of this type would be a busy web server farm. This is made worse by the fact that to deal with clients across the Internet can require more complex network protocols than for communicating within the cluster. Each node may have to use a link on the internal network in order to reach the external network, so this may also heavily stress the internal network.

Unfortunately, in the real world it is rarely that clear cut and the problems solved on clusters may fall into more than one of the above categories. For example, a single web request may result in a composite transaction, with many internal
cluster requests and responses to and from databases and disks. It is also possible for the cluster to move from one category to another as its workload changes.

It is clear that the role of the cluster network will affect these categories in differing amounts, but reducing the cost of communication will have a positive impact on all of them. This work focuses on the network bound cases, but the study of load balancing in Chapter 5 is concerned with all of them.

2.3 Sources of Network Overhead

Network overhead (the cost of transmitting and receiving data via the network) can be attributed to a number of factors. This section examines the effect and contribution made by a number of the components of the communication system. These can, on the whole, be grouped together as: (i) the hardware and its interface to the operating system; (ii) the network protocols; and (iii) the interface between the operating system and the application.

2.3.1 Network Interface

The lowest level of the network is the physical hardware. Although the details of the hardware are out of the scope of this work it is still important to understand how it is accessed.

Physical Layer Protocols

Different types of networks will, at the physical layer, have different inherent efficiencies. This is due to the ways in which they specify that the network medium is accessed (for example through time slicing, tokens, or collision detection) and portions of the raw bandwidth that are used for physical layer headers and data other than those which are passed from the layer above. These differences are discussed for a number of cluster interconnects in Section 2.5.
2.3. Sources of Network Overhead

**Hardware/Software Interface**

The choice of hardware will have an impact through the interfaces it presents to the software built upon it. At the simplest level, the network hardware/software interface needs to be able to support two basic operations:\(^3\):

- Notify the software that new data have arrived from the network.
- Be notified by the software when new data are ready to be sent to the network.

For the former there are a number of different methods available:

*Interrupts* can be raised by the hardware when a packet is delivered from the network to alert the software that there are pending data. This invokes an interrupt handler in the operating system which typically schedules an interrupt bottom half. This results in the application being notified in a timely manner (even if it is not being executed when the interrupt occurs), as the interrupts run at high priority and displace any other tasks that are using the processor.

However, there are a number of undesirable side effects to interrupts. Each time an interrupt is raised the computer must:

- flush the processor pipeline, ensuring all instructions are either completely executed or completely unexecuted;
- perform a context switch (saving the state of the current process and installing the state of the kernel process);
- disable interrupts;
- invoke the interrupt handler to determine what action to take;
- enable interrupts;
- perform any necessary bottom half processing;
- schedule another process.

\(^3\)These are, of course, often supplemented with other operations such as connection setup, signalling, etc.
2.3. Sources of Network Overhead

This causes a considerable amount of computation time to be lost and reduces the efficiency of other parts of the computer (e.g. the cache, the processor pipeline). Although exact measurements will depend heavily on the hardware and drivers in use, interrupt overhead on modern servers running Linux is of the order of 10-20 $\mu$s.

Polling, put simply, is the querying of the hardware by the software. In order to determine when there are packets waiting for it, the software will regularly ask the hardware if any have arrived. If they have, it will deal with them, if not, it will wait a short while before asking again.

Each polling operation is much simpler than each interrupt as it can usually be done without changing context or invoking special handlers. As a result it does not impact on caches and pipelines in the same way. However, in order for the software to be notified in a timely manner it must ask frequently, so there could be many more polling operations than interrupts for each arriving packet. If a lot of the polls find that there is nothing to be done, then a lot of time will have been wasted.

In heavily loaded systems the overheads of servicing interrupts can overwhelm the computer. In lightly loaded systems the overheads of continually polling when there is no work to be done can be similarly disruptive.

For the second operation of the hardware/software interface (notifying the hardware when there are data available for transmission) the method chosen is dependent on the network in question. A number of different networks and the interfaces they provide are described in Section 2.5. Many of these also provide alternatives to interrupts and polling in order to address some of the problems described here.

2.3.2 Protocols

The implementation and requirements of the protocols that allow different nodes in the network to communicate will inevitably lead to some additional overhead.

Protocols define the form of communication between nodes on a network. There is usually a stack of different protocols, each layered on top of another and
2.3. Sources of Network Overhead

each providing different functionality. Layering of the protocols in this way allows
different operations to be decoupled from each other (semantically isolated). The
layers can be easily configured to provide the set of functionality that is required
and one layer can be used by many others, thus promoting code reuse.

While at first glance this is an excellent idea it does have some detrimental
side effects. Although splitting the system as a whole into separate protocols is a
good from a design perspective, when it comes to implementation a different way
of modularising may be more suitable. This is discussed in detail by Clark and
Tennenhouse in their work on Application Level Framing [20]. They demonstrate
how a strictly layered approach prevents efficiency, particularly when implement-
ing transport, session and presentation layer protocols.

The key protocols of relevance to this work are IP, UDP, TCP and the Posix
1003.1g (Berkley-derived) sockets API. A brief overview of them is given here,
but the reader should be aware that many of the finer points and details of these
protocols have been omitted.

**IP**

The Internet Protocol (IP)\(^4\) is the basis for the TCP/IP protocol suite, including
UDP, ICMP and others. It is unreliable (provides no guarantees about successful
delivery), connectionless (each datagram is treated independently, with no initial
handshake phase) and does not guarantee in-order delivery, or even that datagrams
will not be delivered more than once. It is a best-effort service and should some-
thing go wrong data are discarded. Its main function is that of addressing and
routing.

IP came about from a need to interconnect a number of different networks,
using different hardware, all under the control of different institutions. It provides
a common language that enables all devices on the Internet to communicate with
each other.

Each IP packet consists of an IP header (containing for example addressing
information), optionally followed by IP options and some data (the payload). The
header also contains a checksum, performed over the IP header only. (i.e. *Not*

\(^4\)In general, when referring to IP, it should be taken to mean IPv4. This is done for the sake of
simplicity and does not affect the arguments presented.
the payload.) This allows the stack that receives the IP datagram to verify that the header has not been modified. If there is a checksum error, the packet is silently discarded. The transmission of these headers and the computation to compute and check them (in particular the checksum) contributes to the network overhead. The main functionality of IP, namely routing, will also be a source of overhead at each hop as packets traverse the Internet.

**UDP**

The User Datagram Protocol (UDP) is a simple datagram transport-layer protocol. Each UDP datagram is encapsulated (as an IP payload) in an IP packet before being transmitted on the network. The functions it provides to supplement IP are port numbers and a checksum over the payload.

Each UDP datagram contains a UDP header, followed by its payload. The header includes the port numbers (source and destination) which identify the sending and receiving processes. This allows many processes to use the network simultaneously.

The checksum is performed over the UDP header and (unlike the IP checksum) the payload\(^5\). It is designed to catch any changes between the sender and receiver to the UDP header or payload. If there is a checksum error, the datagram is (as with IP) silently discarded. For this reason, UDP should not be considered reliable — any data that are delivered have passed the checksum and so are likely to be correct, but it does not guarantee that all data transmitted will reach the receiver.

In addition, UDP can support broadcast and multicast styles of communication, where more than two nodes are involved.

**TCP**

The Transmission Control Protocol (TCP) is an alternative transport-layer protocol. It provides a very different service to UDP: it is a connection-oriented, reliable, byte-stream protocol.

“Connection-oriented” means that there is a handshake to establish a channel for communication before any data can be sent and similarly to close the channel.

\(^5\)In addition, it includes some fields from the IP header.
once all the data have been received.

TCP provides reliable (and in-order) delivery mainly through its use of a sequence number and acknowledgements. Each TCP header contains (in addition to port numbers and a checksum as with UDP) a number which identifies the byte index in the stream from transmitter to receiver that the first payload byte in this “segment” represents. By combining the sequence number with the length of the segment, the receiving TCP can reconstruct the byte stream, even if the IP data-grams that encapsulate it are delivered out of order. Each time a TCP receives a datagram it replies with an “acknowledgement” to the transmitter. This provides feedback about which packets have been successfully delivered and allows the transmitter to deduce which segments have reached their destination. If a segment is not acknowledged within a certain time, the transmitter assumes it has been lost and retransmits it.

TCP also tries to avoid overloading any point in the network through the use of flow control and congestion avoidance.

Flow control prevents a fast transmitter from inundating a slow receiver with packets. These would quickly consume the finite buffer space the receiver has and result in packets being dropped. Instead, the receiver informs the transmitter (through any packets such as acknowledgements or data travelling in the opposite direction) how much buffer space it has available. This is the window size. The transmitter is not allowed have more than this number of bytes “in-flight” (i.e. yet to be acknowledged) to the receiver at any one time. Once it has sent this many it (and so reduced the available window size to zero) it must wait until the receiver allows it to continue by advertising more available space.

Congestion avoidance [47] uses a similar (window-based) algorithm, but this time the window is imposed by the transmitter and is a measure of the perceived network congestion. The congestion avoidance algorithm assumes that any packet loss in the network is due to congestion, rather than corruption of data, or some other cause. The congestion window is initialised to 1 segment and increases each time new data are acknowledged.\footnote{The rate of increase depends on whether the connection is in “slow start” or not.} When packet loss is detected the congestion window is decreased\footnote{Again, the size of the change depends on a number of factors — the fast retransmit function} and so reduces the rate of transmission. By backing off in
2.3. Sources of Network Overhead

this way it is hoped that the network will stop being congested. This algorithm is far from perfect however, particularly when dealing with high bandwidth-delay product links [50]. A number of alternative ways of dealing with congestion are covered in Section 2.4.2.

Although TCP offers many more features and so seems to be a much better choice than UDP, it is considerably more complex (and so higher overhead) and some of the functions it provides are unnecessary for certain applications.

2.3.3 Sockets API and the Application/OS Interface

The most common way for applications to access the network is the IEEE Posix standard 1003.1g [44] “sockets API”. This is described in detail by Stevens [94]. It connects protocols such as UDP and TCP to the application through a single generic interface. It is therefore touched by issues that are both specific to the sockets API, and those that apply more widely to the application/OS boundary.

One of the most pertinent issues relating to the sockets API is that of its copying and buffering semantics, which are partly mandated by the fact that it straddles the boundary between application and operating system and so is at the point at which “ownership” of data switches between user and kernel space.

To cope with this change in ownership most implementations copy data from a user-space buffer to a kernel buffer (and vice versa). Brustoloni and Steenkiste discuss the effects of this and present some alternative copy semantics [16]. In particular, on the receive path, the application provides buffers into which it would like the data to be placed. This aspect of the sockets API strongly favours a copy, as the data will not have been placed in this final location by the NIC. If particular timing constraints (e.g. sockets API receive function is called before the data are delivered by the NIC) then optimisations such as VMMC-2’s [34] transfer redirection may be possible. Similar techniques can be used when the NIC has on-board buffers that can hold received packets until the application posts a receive buffer; the data can then be delivered directly from the NIC to the application space [25]. If buffer size and alignment constraints can be met then page table

---

8 Throughout this text this standard (and conforming implementations) are referred to as “the sockets API”.

---

for example.
2.3. Sources of Network Overhead

Manipulations may be possible to avoid copying [33, 40], but in general these result in a loss of flexibility and will require modifications to applications to meet the constraints: they effectively change (through restrictions) the API. Finally, rather than avoid this copy, the cost can be amortised by performing it in parallel with other actions that must touch the data [92], such as checksumming.

The boundary between applications and the operating system kernel also contributes to network overhead through the complexity of the “house-keeping” operations associated with a system call and context switch. When moving from the restricted user domain to the unrestricted and privileged kernel domain, the operating system\(^9\) must:

- Intercept the call into the OS and execute the system call handler;
- Enable access to the protected kernel regions of memory;
- Save the state of the application and the user process stack;
- Instantiate the kernel stack;
- Perform any checks on mutual exclusion for access to the kernel that are necessary;
- Invoke the function that was requested.

There will also be a significant amount of parameter checking to ensure that the kernel doesn’t inadvertently perform an illegal operation. On exit from the system call, the reverse of the above operations must be performed.

The costs of performing a system call can be measured by timing an operation such as `getpid()`. Muir and Smith, when discussing their Piglet architecture [64], show this to be of the order of a few microseconds for a Unix system.\(^{10}\)

---

\(^9\)The exact sequence of operations will depend on the type of operating system in use. For example macro- or micro-kernel.

\(^{10}\)The exact time will be heavily dependent on the OS and hardware architecture in use.
2.4 Dealing with Overload

All servers, whether through inefficiency (i.e. high network overhead) or through simply being asked to do more work than they can handle, risk becoming overloaded. This section examines possible symptoms of overload and approaches to avoid or mitigate it.

2.4.1 Receive Livelock

One of the most important concerns for a server is that of receive livelock. This is the condition where a server becomes overloaded due to the rate of packets being delivered to it. When each packet is delivered some work must be expended by the server. Typically the NIC will raise an interrupt notifying the operating system that a packet has arrived. The interrupt service routine is run at high priority. If the time interval between packets arriving at the server is less than the time taken to handle the interrupt, the server will spend 100% of its time servicing the network. In this state the server does no useful work because there are no CPU cycles left to process the data. As a result partially processed packets are continually dropped (wasting the work that has already been done on them) to make way for new ones. This is because the initial processing takes place at higher priority than later processing and delivery.

It is possible to combat receive livelock by reducing the amount of time it takes to deal with each packet. An approach known as interrupt hold off (or interrupt coalescence) is widely used in current servers to do just this. When a packet arrives the NIC does not immediately raise an interrupt. Instead it waits for a short length of time in the expectation that another packet is likely to arrive. If more packets do arrive soon after then the cost of the interrupt is amortised across all of them. This comes at the cost of increasing the average latency and also requires the NIC to be able to buffer a number of packets. Increasing the size of the packets on the network has a similar effect. To transfer a fixed amount of data requires fewer packets and so fewer interrupts. However, to form large packets

\[\text{If for example interrupts are raised on every packet on a Gbps network and the packets are 128 Bytes long, there will be over one million interrupts each second. If each interrupt takes 10 \mu s, this corresponds to 10 seconds of work.}\]
2.4. Dealing with Overload

may require the delaying of data at the sending node and so again impact latency. Both of these techniques also only postpone the problem to even higher packet rates.

Mogul and Ramakrishnan [48] propose a more robust approach to receive livelock. By using interrupts only when the server is lightly loaded they retain responsiveness when the server is idle without the overhead of continually polling the network. After an interrupt the operating system switches into a polling mode, in the hope that another packet is likely to arrive soon after the first. By using polling in this way, as the packet rate increases the large hit of an interrupt per packet is avoided. They also allow the system to drop packets very early when the server becomes loaded. In this way they avoid processing data that may be dropped at some later stage. Druschel and Banga [32] take this concept further. By having early demultiplexing into per-application queues rather than a single IP queue they can process the packets later at the priority of the application. This “Lazy Receive Processing” allows them to drop packets early (if the application does not process them) and achieve better separation and fairness between packets for different applications. As the protocol processing is only performed when requested by the application, the active (at the time of delivery) process does not get removed from the processor to service the network when it was able to make progress with its own work. This reduces switching between contexts and promotes cache-efficiency.

It is also possible that hardware can help with the problem of receive livelock. NICs can be classified [9] as self-selecting or non-self-selecting. A self-selecting NIC is able to demultiplex a stream of incoming packets to its (possibly final) destination in memory. A non-self-selecting NIC leaves this job to the device driver. Clearly the self-selecting NIC will reduce the load on the operating system. It does however require more sophisticated hardware. There is a risk that the hardware itself could become overloaded (depending on the capabilities of the hardware) and so the problem could remain.

The problem of receive livelock essentially stems from the situation where there is a fast transmitter (or many slow transmitters) inundating a server with packets faster than it can deal with them. The fundamental solution is to give the server control over how fast the transmitters can send data. By only allowing the
transmitters to send data when the receiver has explicitly given them permission to do so, the receiver can be sure that it will not receive more packets than it can cope with. One method for implementing this is credit-based flow control [52]. This also happens to be a side effect of many send-directed networks.

A send-directed network is one where the transmitter, rather than the receiver, specifies where in the receiver’s memory a particular packet is placed. To do this the receiver must inform the transmitter which portions of memory it can use, thus giving it permission to transmit a certain amount of data. Obviously, once it has written to all the permitted buffers, the sender must wait to be given more “credit” in the form of permission from the receiver to use other areas of memory. CLAN (discussed in Section 2.5.8) is a good example of this.

2.4.2 Congestion Control and Checksums

In any inter-network path there will be one link or device which forms a bottleneck. It will either be running slower than the rest of the network or form a part of a popular route. At peak times this point in the network is likely to reach the limit of its performance and become overloaded.

There are techniques to deal with network overload at many different layers in the network:

At the physical layer for example, Ethernet forms one of the most popular technologies for local area networks. The Ethernet MAC uses collision detection with an exponential back-off (this algorithm is known as CSMA/CD) to deal with the case where two nodes on the same network link try to use it at the same time. This works well when the probability of a collision is low (i.e. at low loads) but as the load rises the number of collisions increases and so the utilisation of the Ethernet sharply decreases. This weakness has been typically addressed by partitioning the network to ensure that each link has a small number of hosts and then using more complex switching equipment to aggregate and connect those links to form a network.\(^{12}\) Over-provisioning of bandwidth in the local area has also been both popular and successful, albeit expensive. More sophisticated token-based systems

\(^{12}\)In fact, CSMA/CD is rarely seen in use any more; Ethernet has become a fully switched network.
have existed for many years, but despite their improved performance at high loads the flexibility of Ethernet has made it more popular.

Rather than using the physical layer, which may only be able to control the congestion over a single hop, end-to-end protocols such as IP can address the problem. Active Queue Management (AQM) and other schemes allow central network equipment such as IP routers to deal with congestion. There are many algorithms for AQM (for example RED [38]), but they all specify how to deal with queues of packets that build up within congested core network equipment and how to keep the queues small. AQM has however been shown [54] to have little effect unless coupled with Explicit Congestion Notification (ECN).

ECN [73] attempts to provide feedback to the sender of data about congestion in the network using bits in the IP header. This allows the sender to reduce the transmission rate and so hopefully allow the network to recover. It is an important step forward: without it packet loss is used by protocols such as TCP as the main indication of congestion, but packet loss can be attributed to a number of other causes. This can often lead to the false triggering of a host’s mechanisms for dealing with congestion within the network when it is unnecessary and this can have a large impact on performance (as congestion recovery can be an expensive operation).

Although built on top of IP, TCP was itself developed in response to congestion collapse of network links. It includes a mechanism to find and maintain a transmission rate that does not overload the network. (This algorithm was outlined in Section 2.3.2.) Before the advent of TCP in the early days of the Internet this role was left to applications. Some applications (typically those that use something other than TCP) will still take responsibility for this.

**End-to-End Reliability**

Related to this discussion of congestion control is one of reliability. Congestion is one of the major causes of loss or corruption of data in a network link. TCP achieves reliability by the fact that it operates “end-to-end”; i.e. the checksum is computed by one end of a connection and checked by the other. It is not, in most cases, calculated by intermediate nodes. If there were a portion of the path that
2.4. Dealing with Overload

was not protected by the checksum (even if only within one intermediate router) any errors introduced at that point would go undetected.

In addition to the end-to-end TCP checksum, a per-hop CRC is often provided (depending on the physical layer being used). However, for the above reasons, this does not provide an adequate replacement for the end-to-end checksum. Stone and Partridge have shown that relying on link-layer CRCs alone would be dangerous and how even the combination of CRCs and end-to-end checksums does not provide infallibility [95].

This is particularly worrying when considering the current trend for off-loading portions of TCP (such as the checksum computation) from the end node, or terminating TCP connections at a “front end” and using another (lower overhead) protocol within a cluster. These approaches require the assumption that the cluster network is reliable, which may be possible, but is often not the case.

Checksums therefore remain, although a large source of overhead, very necessary and careful thought must be given to their implementation to ensure correct behaviour and reliable data delivery.

Cluster-Network Congestion

Although there exists much research into dealing with the problem of congestion when it occurs in the core network, this work is most interested in what happens when it is a component of the server cluster (either the local network or one of the nodes) which becomes overloaded. It would be possible to take a similar approach to the core network solutions above and provide feedback from the server cluster to the clients in order to control the load. However, as the server cluster is typically distant from the clients and can suffer very transient loads due to serving a large number of clients at once, the time constant of the feedback loop can be too large to be useful (or even stable). TCP and application-level approaches to congestion are of some help but do not represent a complete solution. In some senses they address a different problem, so are complementary. As a result, clusters of servers need some way to address to control the situation locally. This is the concept of load balancing.
2.4.3 Load Balancing

Load balancing is the process by which a cluster divides the work it is asked to do among the available nodes to ensure that no one node receives a disproportionate fraction of the workload. Approaches to load balancing can be categorised as follows:

**Centralised:** A single node in the cluster is responsible for dividing the work among the other nodes. This node is often a gateway into the cluster and commonly referred to as the *distributor*. Often the cluster “advertises” a single IP address so it is completely transparent to the client which server node actually deals with its request. The distributor can use a number of different policies to decide which node should perform the next unit of work.

The simplest of these is round-robin scheme. The distributor simply distributes incoming requests to the servers in turn, regardless of the complexity of each request or any differences between the servers. This works well in a cluster where the servers are homogeneous and the units of work all represent a similar amount of computation. When this is not the case it can be improved by using a weighted round-robin scheme. Here a weight is assigned to each node to reflect its abilities and these weights alter the frequency with which that node is chosen.

A number of different metrics (CPU usage, number of outstanding requests or concurrent connections for example) can be used to measure the current load of each node. Using the number of current connections to the node (possibly together with some measure of the level of activity on that connection) is attractive as it is already known by the distributor. It does require the distributor to maintain state about each node, which can be costly. Centralised systems also suffer from the distributor being a single point of failure or a bottleneck.

Attempts to reduce the bottleneck of the distributor include only using the distributor to dispatch connections, leaving the server nodes to communicate results and any further negotiation directly with the client [42].

**Distributed:** The nodes use some means of passing off requests to other nodes
2.4. Dealing with Overload

when they are heavily loaded (or requesting work from other nodes when they are lightly loaded). These are generally more complex than centralised approaches, but can be more scalable.

As before, the simplest method (for an overloaded node) when deciding who to pass some of its load to is just to pick one at random. This works tolerably well for lightly loaded clusters where the probability that the random node will be less loaded than overloaded node is high.

If this is not the case there must be some way to discover the loads on other nodes in the cluster. A distributed system does not have the luxury of explicit knowledge about the other nodes as the distributor did. Load information can either be obtained using a periodic broadcast, or by polling other nodes.

If a periodic broadcast is used each server must measure its load and broadcast this information to the other servers. For this information to be useful the broadcast frequency must be high: if too low it runs the risk of the information being stale and out of date. However, as the broadcast frequency is increased, the overhead of gathering the load information also increases. The cluster may also exhibit a flocking behaviour where work swiftly congregates on the server with the lowest load, immediately overloading it.

If polling is used it avoids the risk of the information being stale (assuming the latency of the network is low enough) as the information is requested on demand. However, it can result in additional delay while the node waits for the response to its load query. It is also necessary to decide how many other nodes in the cluster to ask. This is a trade-off between the overhead of gathering load information and the accuracy of the decision. Research has shown that a polling set size of as little as 2 or 3 (in a cluster of 16 nodes) can yield good results, while increasing the set size gives little extra benefit [88].

A distributed system also requires some method for the efficient hand-off of jobs from one node to the other. Distributed Packet Rewriting [7] is one example of this.
A further aspect of load balancing is locality-aware [67] request distribution. This aims to distribute the requests more intelligently than any of the above examples. Sending the requests for the same data (when the load permits it) to the same node in the cluster, means each node should see improved locality and cache performance and also allows for the use of more specialised nodes. This approach is more complex, particularly for the distributor which must examine the incoming requests in detail to determine which node to forward them to. This leads to issues of scalability [6]. It must also cope with the case where one node is becoming overloaded with requests for a single object.

It is also necessary to consider the level at which load balancing takes place. Most distributors work at the transport layer and distribute load at the granularity of TCP connections. Others work at the application layer and try to provide more fine-grained load balancing. For example, with persistent HTTP [60] a single TCP connection can contain many HTTP requests and it may desirable to distribute each of these to a different node, rather than have one node deal with the whole TCP connection. This is discussed fully, together with a in-depth discussion of various commercially available load balancers, in a paper describing Cyclone [90]: a cluster-based web server that advocates balancing at the level of the HTTP request.

2.5 Networks for Server Clusters

There are a number of choices of commercially available interconnects which are suitable for a server cluster [76]. The decision about which is most suitable will depend on many factors as described in Section 2.1. There have also been an number of non-commercial research projects that have produced networks of interest. This section provides a broad overview of these technologies and some of the trade-offs that are involved, but first outlines some of the common concepts.
2.5. Networks for Server Clusters

2.5.1 High-performance networking concepts

User-level networking

Many of the networks discussed here achieve their high performance by virtue of being accessible from user-level. Traditionally network access has been performed on the behalf of applications by the kernel: this is kernel-level networking. User-level networking is the transfer of data directly to and from the network from an application, without the direct (or sometimes partial) involvement of the kernel. This is discussed in detail by Riddoch in his Ph.D. thesis [78].

RDMA

RDMA [8] (Remote Direct Memory Access) is an extension of the intra-computer concept of DMA. DMA is the transfer of data from one component (such as a disk-drive) to another component (such as RAM) within a PC without the direct involvement of the CPU. The CPU simply initiates the transfer, but it does not read and write all the data. RDMA extends this to networking and operates between computers. It allows one computer on the network to transfer data from a component within it, to a component in another computer, without the direct involvement of either CPU. Again, the CPUs initiate the transfer, but do not read and write the data themselves.

2.5.2 Gigabit Ethernet

Ethernet has for many years been the most popular networking technology. Its flexibility and ease of deployment, along with cheap hardware, have resulted in it becoming dominant in the area of wired LANs despite a number of weaknesses (for example those described in Section 2.4.2).

In the past, Ethernet was not seen as a viable network for high-performance computing, due to its relatively low bandwidth, poor performance at high loads, its media access control (MAC) protocol did not scale well to large numbers of nodes and it was not able to provide any guarantees about the quality of service (QoS). However, order of magnitude increases in bandwidth, a move toward switched networks and improvements to the MAC along with over-provisioning the net-
work (to mitigate the QoS issue) have resulted in it being a serious contender in high-performance networking.

Ethernet has also managed to retain its market share by offering an easy upgrade path as the bandwidth has increased. Gigabit Ethernet is currently commercially available, with 10 Gigabit Ethernet undergoing the standardisation process. As most servers will ship with at least one Gigabit Ethernet NIC people are familiar with its performance. As a result it forms an important reference point for other technologies to compare against.

Much work has been done to make Ethernet perform well at gigabit speeds (and beyond) and a number of ways of addressing some of its problems (such as “jumbo frames” and interrupt coalescing) have been developed.

Level 5 Networks [56] is a company whose products demonstrate that Gigabit Ethernet can be a high-performance network. They have developed a 1 Gbps Ethernet-based network called EtherFabric. This is able to use similar approaches to networking as the CLAN project (described in Section 2.5.8) while remaining compatible with other Gigabit Ethernet hosts.

### 2.5.3 Myrinet

Myrinet [12] is a multicomputer (ie. aimed at a cluster) message passing network. Myrinet consists of specialist NICs, switches and cabling. The NICs provide a processor which is able to provide some offload and network processing operations and provides a user-level accessible interface. Myrinet is very flexible, allowing the firmware on the NICs to be reprogrammed for a particular optimisation. However, all communicating nodes must be programmed in the same way.

Many research projects have made use of Myrinet networks. These include Trapeze [110] and the SHRIMP VMMC-2 [34] which is outlined in Section 2.5.7.

Trapeze is a messaging system for the Myrinet network. It is of particular interest to this dissertation as a zero-copy TCP [40] has been developed that uses Trapeze and Myrinet. This is discussed in more detail in Section 2.6. Trapeze, as with many other networks, allows the host to communicate with the NIC through the use of two message rings (one for sending, one for receiving). These contain 128 byte messages, with an optional payload attached. This enables it to separate
headers from payloads and (combined with Myrinet’s DMA capabilities) perform scatter/gather operations. Payloads are transferred to and from aligned pages of host memory. It is also able to support very large transfers, as Myrinet does not impose an MTU. An unusual ability of this network is the overlapping of send and receive DMA transfers with network access, as opposed to the normal store-and-forward technique, as illustrated\textsuperscript{13} in Figure 2.2.

![Figure 2.2: Store-and-Forward vs Cut-Through](image)

2.5.4 VIA and U-Net

The Virtual Interface Architecture (VIA) [35] was an industry standard which built on academic work done on the U-Net project. U-Net [105] was developed as a low latency user-level message passing network. It uses ATM hardware, with a co-processor on the NIC which is accessible from user-level.

The application is able to access the network using a number of different queues. To transmit data the application places the data in a buffer and creates a descriptor that describes the buffer and the destination. This descriptor is then placed on the send queue and will be serviced by the NIC. The application must also place descriptors on a free queue to describe buffers into which data can be delivered. As the NIC fills these it places them on the receive queue. To receive notification the application can poll the receive queue, use the OS `select()` mechanism, or register a callback.

Extensions to the base network have been developed to allow transfers from

\textsuperscript{13}This illustration is taken from [110].
application buffers that can be paged out [107], rather than from a special block of “pinned” memory. This is achieved by adding TLB functionality to the NIC, which is kept in sync with the main OS TLB. This allows for an RDMA write facility where the sender is able to specify where, in the address space of the receiver, the payload should be delivered.

To simplify transfers of small messages (which are often the most latency sensitive) the network allows their payload to be included in the descriptor. This reduces the overhead of buffer management.

VIA uses a similar communication model to U-Net. It has the same queue system and in addition to RDMA writes it also provides RDMA reads. It is connection oriented, meaning that descriptors in the send queue do not need to identify the destination, as this is explicitly defined by the connection. In VIA each descriptor may contain a number of data segments which together form a single message, allowing scatter/gather. VIA uses completion queues to efficiently notify the application about changes to the send or receive queue. When a descriptor has been processed the Virtual Interface (VI) will post notification to the relevant completion queue. The application can either poll or block on these queues.

VIA supports two different qualities of service. Unreliable delivery provides no guarantees about loss or order of delivery, but does provide error detection and will deliver each message at most once. Reliable delivery guarantees that messages are delivered in order with no loss. Should loss occur the connection is closed to provide this guarantee. VIA does not provide flow control at the application level, so loss can occur as a result of the receiver not providing enough buffers for received data. In the unreliable class any over-running messages would just be dropped, but in the reliable class the connection would be closed. It is therefore necessary for applications making use of the reliable class to either hope the receiver can keep up, or implement its own flow control mechanism.

Although VIA is now no longer developed commercially, there were a number of research projects that implemented the VIA API over a variety of different hardware platforms [91, 80, 17] and it has formed the basis for its successor: Infiniband.
2.5. Networks for Server Clusters

2.5.5 Infiniband

The Infiniband Architecture specifies a switched fabric System Area Network (SAN). It has been developed by the Infiniband Trade Association (which consists of a number of large companies in the network industry). With currently available silicon supporting 10Gbps data transfer rates, it is targeted at the data centre. It is seen as complimentary to technologies such as Gigabit Ethernet, which (the Infiniband Trade Association suggests) should be deployed at the edge of an Infiniband network.

Infiniband distinguishes between processing nodes and I/O nodes in the network and defines two channel adaptors\textsuperscript{14} that can be used to connect to the fabric.

The Infiniband Architecture defines a system of queues for the application to communicate with the channel adaptor much like VIA. There is a send queue on which requests to transmit data can be placed, a receive queue to contain buffers that can take incoming data and a completion queue for notifications about send, receive and other events.

The Infiniband network supports both channel-based semantics and memory-based semantics. In the channel-based mode, the queues are used as above and the network is receive directed; i.e. the receiver decides where to put the data. It does so by posting buffers to the receive queue. In the memory-based mode the network supports RDMA operations and so is send-directed. It also supports a variation called “RDMA write with immediate data” where a small block of data is passed along with the RDMA and this data block is placed on the receiver’s completion queue.

There is no standard API for accessing Infiniband hardware. The specification instead describes the functionality that the channel adaptor provides: “Verbs”. This can result in different manufacturer’s channel adaptors having different APIs, despite providing the same functionality. Verbs does not provide any facility for flow control and so this is left to protocol layers built on top of it. If a message arrives at its destination and there are no suitable receive buffers, a retry mechanism is invoked. While this is better than VIA’s approach (of dropping the packet or closing the connection) it can result in a large performance degradation.

\textsuperscript{14}A channel adaptor is notionally equivalent to a NIC in this context
Buffer management is also a costly operation. The send and receive buffers must be in pinned memory and registered with the NIC: an operation that involves the application, the operating system kernel, and the NIC. The size of the buffers must also be chosen carefully to ensure that they are (individually) large enough to accept any transfer as a single transfer can not be split during delivery across multiple receive buffers.

### 2.5.6 SCI

The Scalable Coherent Interface (SCI) was originally developed as a high performance bus for interconnecting processors in a multicomputer. It is based on distributed shared memory and includes considerable support for cache coherency.

As cluster computing became more popular SCI was adapted to be used as a PCI card and so enable nodes in the cluster to communicate. However, as PCI cards are not able to participate in transactions on the main bus (between the CPU and memory for example) coherency is lost. A variety of protocols have been developed to support cluster communication over SCI [83], including shared-memory protocols, message passing protocols and TCP/IP [96]. Many of these allow user-level networking.

The raw network is unique among those discussed here in that it is parallel (16 bit wide bus) rather than serial. As a result it has a theoretical bandwidth of 8 Gbps. Due to the low overhead of distributed shared memory some of the protocols are able to deliver latencies of as little as 5 $\mu$s, with bandwidth peaking around 660 Mbps.\(^{15}\)

Synchronisation is a problem for many shared memory interconnects, including SCI. As the PCI card is able to transfer data to and from the host’s memory without involving the host’s CPU, the CPU is not automatically notified that a transfer has taken place. There is, on the whole, a choice between polling the memory or raising an interrupt, but these have the problems outlined in Section 2.3.1.

\(^{15}\)It is not clear from [83] whether or not these results are application to application measurements, or if the applications touch the data, making it difficult to compare to other published results.
2.5.7 SHRIMP VMMC

SHRIMP is a multicomputer project at Princeton. Virtual Memory Mapped Communication (VMMC) is the network technology they have developed for SHRIMP. It is aimed at a similar market to SCI.

The initial implementation of VMMC [11] used custom hardware. It supports communication by mapping physical memory from one node to physical memory in another. Updates to the local memory are forwarded to the remote memory, which earns it the name reflective memory. To communicate with the NIC the application uses a command page. Each page of mapped physical memory has associated with it an extra page of virtual memory (the command page) which in turn is mapped onto the NIC. Writes to the command page are used to trigger DMA transfers across the network from the corresponding mapped physical page. This allows DMAs to be initiated using a single instruction, but does result in a large amount of virtual memory being consumed.

A second generation network interface VMMC-2 [34] has also been developed. This is based on Myrinet hardware. It introduces a number of features to improve the flexibility of the placement of memory buffers. For example, it no longer requires a fixed mapping between the send and receive buffers. The NIC and driver maintain a TLB of pages of application memory that are accessible by the NIC. This TLB is cached on the NIC to prevent it having to access host memory on every lookup. This allows it to support transfer redirection: if a receive operation is initiated before the corresponding data are delivered, the NIC is able to redirect the incoming data to the buffer the application describes as part of the receive, so preventing it having to copy the data (at the receiver) from a temporary buffer to the final destination. VMMC-2 also provides the option of a reliable data-link layer, allowing it to support the sockets API directly, without the need for TCP/IP. When compared to the sockets API built on TCP/IP over Myrinet this unsurprisingly delivers a dramatic increase in performance (bandwidth of 670Mbps and latency of 20us).

One of the weaknesses of SCI was that interrupts must be raised on every transfer in case the receiver was blocked. SHRIMP VMMC-2 is able to improve on this by allowing the application to request that an interrupt be raised whenever
2.5. Networks for Server Clusters

a particular page of memory is accessed. This allows the receiving application to indicate when interrupts are necessary and so increase the efficiency of synchronisation. It is still relatively coarse grained however, as it is unable to distinguish between many different updates to the same page (which is likely to be a common access pattern) some of which the application will not need to be immediately notified of.

2.5.8 CLAN

CLAN is a low-latency, high-performance, user-level network developed at AT&T Laboratories — Cambridge Ltd. Its primary targets are the server room and cluster computing.

In previous work Riddoch et al have published detailed descriptions of the hardware and software support, as well as Tripwire [81] (the synchronisation mechanism and its use to support a variety of protocols and applications [79] including VIA [80].

CLAN Hardware

There is a prototype hardware implementation consisting of a number of Network Interface Cards (NICs) and two 5-port switches. A bridge (between CLAN and Gigabit Ethernet) was in development. The current hardware has some weaknesses (for example the DMA engine only allows a single request at once and generates an interrupt after each transfer) but represents a viable platform for research into software support for user-level networking.

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

The bridge was still under development when AT&T Laboratories - Cambridge Ltd closed in April 2002. As a result, it is currently unfinished. To allow bridging experiments to continue a prototype bridge consisting of an dual processor server PC equipped with CLAN and Gigabit Ethernet NICs has been used to perform this role. A new version of the NICs designed to run at 3 Gbps was also in the pipeline when the laboratory closed.
2.5. Networks for Server Clusters

Low-Level Data Transfer

At the lowest level CLAN is a Distributed Shared Memory (DSM) interface. Regions of physical memory (called apertures) can be mapped from one host across the network to the virtual memory space of an application running on another host. The same region of memory is also mapped into the address space of a local application. This allows the two application to communicate through very low latency transfers using standard processor write instructions. In this way it is similar to the SCI network. Unlike SCI however, it does not attempt to address issues of coherency. Instead when data are transferred to a remote aperture, they are not cached locally. This greatly simplifies the underlying protocols, but has the drawback that reading from a remote aperture incurs higher cost.

In addition to Programmed IO (PIO) the CLAN NIC also provides a DMA engine to allow longer transfers to be offloaded from the CPU. This is RDMA, where one node can DMA data to or from another in the network.

An RDMA cookie describes an aperture and can be passed to another host in the network to give it the ability to access that aperture. The prototype network does not however provide any security against malicious code writing to apertures that it should not have access to. This would be essential in a commercial implementation and a solution similar to that developed for Hamlyn [108] could be used.

Although at the network layer the network is not connection oriented, it supports out of band messages that can be used to provide connections and the majority of useful abstractions choose to do this.

The API for CLAN takes a different approach to other user-level networks such as Myrinet [12], U-Net [105] and SCI [83]. The CLAN NIC presents a single, low-level network interface that supports communication with low-overhead and latency, high-bandwidth and efficient and flexible synchronisation. More complex interfaces can be built on top of this without considerable additional overhead. The NIC is implemented using simple hardware without on-board processors.

Although at the lowest level CLAN is a DSM network, it is not intended that the normal DSM style of communication is used by applications. Instead, the
2.5. Networks for Server Clusters

DSM support is used as the base for building higher level communication abstractions.

**Distributed Message Queue**

One of these abstractions is the Distributed Message Queue (DMQ) as shown in Figure 2.3. It is essentially a flow controlled messaging abstraction and is described here in its simplest form. By providing end-to-end flow control the network relieves the application of this task and simplifies the implementation of protocols that use the network.

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

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

Synchronisation is performed using Tripwires [81] which provide a low-overhead mechanism for notifying the application of changes to the queue. This addresses many of the problems that have been experienced with other distributed shared memory networks, where synchronisation can be problematic.

Discussion

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

CLAN’s sender-directed approach of using distributed memory and RDMA to transfer data is becoming increasingly popular, with the IETF standardising aDirect Data Placement (DDP) API [8] and hardware is currently available to do this over Ethernet from companies such as Level 5 Networks.

2.5.9 MemNet

MemNet [30] takes an alternative approach to cluster networking. Like CLAN, it uses distributed shared memory. However, the memory that is shared resides on the NIC, rather than being normal host memory. Each page of shared NIC memory can be cached by other nodes in the network and cache consistency protocols are used to ensure that the multiple copies of the same memory location are kept up
to date. MemNet uses a token-ring network to propagate changes, which has the advantage that each packet on the token-ring is passed to every node in turn, in the same order. Latency suffers, however, in a token-ring network as the number and distribution of nodes increases. The use of NIC-based memory is also a drawback, as it is likely to be more expensive, limited in size and more costly to access when compared to host memory.

2.5.10 Discussion

It is clear from the above outline of a number of user-level technologies that they have the ability to offer increased performance and flexibility over traditional interfaces. However, there are a number of trade-offs to be considered.

The cost of user-level networks is in general higher than traditional ones. This is partly due to their “exotic” nature and the increased performance they can provide resulting in them being high-end cards and so manufacturers can charge a premium for them. However, a large part is simply due to the additional complexity of the network interface cards and other components in the network.

While the networks are able to deliver impressive raw performance (which can be sometimes be passed on to tuned applications using specialist APIs), the interface to the application can often be a bottleneck. This is largely due to the problems of integrating an unconventional network through conventional interfaces (to maintain compatibility with existing applications) while still allowing the application to access other system resources in the normal way.

By placing network access at user level, networking is no longer controlled from a single point (the kernel) and so managing multiple endpoints can be difficult, or even just not supported. This is particularly the case where an application wishes to use a single system call such as `select()` to manage both user-level and normal OS endpoints.

While raw performance using specialist protocols is helpful, it is important that the user-level network can be interfaced to the wider Internet and some of this performance retained. To do this requires the use of standard protocols, such as TCP/IP, as well as all the “control-plane” protocols that go with it such as ARP and ICMP. The following section examines how it is possible for user-level networks
2.6 User-Level TCPs

Placing protocols at user level is not a new concept. This area has been steadily researched for many years and has delivered a number of novel and interesting results. Early techniques often involved a dedicated server process to implement the protocol.\(^\text{16}\) This decoupling of the protocol code from the kernel resulted in greater flexibility, allowing it to be more easily modified and optimised [62, 21, 39].

However, the greater flexibility came at the cost of greater overhead, which inevitably led to poor performance. This was due to the protocol service being executed in a different process to the application and so data travelling between the application and the network (in either direction) must cross extra process boundaries, leading to unnecessary switching, copying and inefficient use of APIs. This is demonstrated by Maeda and Bershard: they assert that the reason for poor performance of user-level protocol implementations is not because it is at user level, but because a user-level stack requires a different architecture to a kernel one [22].

The rest of this section discusses a number of different research projects that have developed this concept of protocols at user level into high-performance implementations.

2.6.1 Protocol Decomposition

Maeda and Bershard highlight the weaknesses in a pure server-based protocol implementation [57]. They suggest that the protocol can be decomposed into a number of components, in particular that the interface to the network is separable from the interface to the operating system. They propose an architecture where the non-performance critical components such as connection setup and tear-down are implemented in an operating system server (at user level) but the send and receive functionality is linked directly with the application through the use of a

\(^{16}\)This is often seen as a micro-kernel approach, where the kernel contains a minimal set of functionality and instead there are a number of user-level services.
multi-threaded library. Both of these communicate with the network through the
kernel, which provides a thin abstraction of the hardware. The protocol library and
operating system server must cooperate and communicate between themselves
using RPC.

Their architecture maintains the flexibility of early approaches, but has the
potential for higher performance and their results demonstrate that it is able to
match the performance of traditional kernel implementations at that time.

Thekkath et al also expound the virtues of a move away from single trusted
user-level servers toward a multiplicity of user-level libraries [99]. This paper
includes an excellent discussion of the motivation for user-level protocol imple-
mentations, the mechanisms required to support them and the disadvantages of
their approach. The architecture they choose to implement is similar to the one
proposed by Maeda and Bershard, but includes a number of optimisations. For ex-
ample, the application and kernel use shared memory to communicate and transfer
packet data. These optimisations allow it, for some transfer patterns, to slightly
outperform kernel implementations (and also show significant performance im-
provements over user-level server implementations).

There have been experiments with other ways of splitting the code between
kernel, user-level library and user-level server. Braun et al [14] for example choose
to implement TCP at user-level, but leave IP and the other components in the
kernel. This unusual approach is motivated by the move toward Application-Level
Framing (ALF) and Integrated Layer Processing (ILP) [20].

However, due to inefficiencies in the implementation when compared with a
well tuned kernel stack it performs relatively poorly. Performance is sacrificed
to provide the flexibility needed for ALF and ILP and it highlights the problems
associated with user-level protocol processing.

2.6.2 Jetstream

The projects discussed above have all used standard networking hardware, using
the kernel to provide an abstraction of it for use from user-level. Jetstream [106]
was among the first to develop specialised hardware for this purpose. Jetstream
is a Gbps token ring network that uses AAL-5 frame format. The NIC is com-
2.6. User-Level TCPs

combined with an expansion card (Afterburner) which provides memory for network buffering. The device driver has a low-level access interface to allow applications to gain direct control over Jetstream functionality.

When considering the design of TCP/IP for Jetstream [36], Edwards and Muir leave much of the code in the kernel. This includes packet demultiplexing (for security reasons) and packet buffering (to allow efficient pool management). They choose to adapt the TCP from the HP-UX 9.01 kernel and port it to user-level. The design of the structure of their user-level stack is also based on the partitioned nature of the kernel, with the protocol processing being performed in a separate process\(^\text{17}\) which communicates with the application process using shared memory. The application process also provides the necessary system calls for the application. The protocol process, although similar in some ways to the user-level server used in some of the above schemes, is only accessible by one application. It is scheduled as a real-time process, giving it similar priority to the kernel bottom-half that it mirrors.

There is a single trusted “connection server” running on each node which deals with connection setup requests and hides the details of the underlying network connections (Virtual Channels).

As the device driver allows the application to gain low-level access to the hardware the stack is able to operate a single-copy mode of operation. This goes someway toward helping the performance of Jetstream TCP/IP.

At the time that Jetstream TCP/IP was developed, kernel stacks were moving from multiple copy to single-copy implementations. The single-copy kernel-based TCP gave a bandwidth of around 200 Mbps, while a standard kernel-based TCP was recorded at approximately 100 Mbps. Jetstream TCP/IP was able to deliver bandwidth that fell between these two: 160 Mbps — less than one fifth of the line speed. It was limited by the CPU usage on the receiving node, which can be attributed to the overhead in communicating via the protocol process (i.e. additional context switches).

\(^{17}\)The authors suggest that using multiple threads in the same process would be beneficial in an environment that supports this.
2.6.3 Nemesis

Nemesis [55] is an operating system developed at the Cambridge University Computer Laboratory. It has a rather unconventional structure. The kernel itself is tiny, consisting mainly of a scheduler and interrupt and trap handlers. It differs to micro-kernel operating systems (which also have very minimalistic kernels) in that the functionality is not moved into servers, but to the application processes instead. The goal of the Nemesis project is to provide QoS. To do this all resources, include virtual ones, are scheduled (rather than just the CPU).

The unusual structure of the operating system leads to some interesting ways of structuring the network protocol stacks [10, 103]. Again, the aim is for QoS rather than performance and so care is taken to ensure there is no crosstalk between applications using the network. It is for this reason that a user-level server would be unsuitable for network access. The user-level server would be a shared resource and so could experience QoS interference between applications.

In the other user-level TCPs discussed some of the functionality remains in the kernel, even if only to handle special cases. Here all protocol code is moved to the application, making this one of the few pure user-level implementations. Each application contains an IP stack for each of its IP flows. Nemesis also supports “user-safe devices” which provide an interface such that the data path can access the hardware without using the device driver.

Nemesis is able to achieve its goals of providing QoS guarantees in terms of network access and delivering much increased fairness between competing flows. It is able to fully utilise a 100Mbps Ethernet.

2.6.4 Trapeze TCP/IP

Trapeze, as discussed in Section 2.5.3, is a Myrinet-based messaging system, implemented over a Gbps network. It is a relatively recent development compared to some of the above so has the advantage of much more sophisticated hardware.

Trapeze TCP/IP focuses on reducing the overhead of moving data. This is a per-byte rather than a per-packet overhead, but is not reduced in line with improvements in CPU speed as it is largely limited by the speed of the memory system.
2.6. User-Level TCPs

To address this Trapeze TCP/IP includes a zero-copy sockets API. This uses page-remapping to transfer pages containing packet data from the kernel to the address space of an application. It requires that packets are page aligned and ideally their length should be an even multiple of the page size. For sending data through the sockets API a copy-on-write scheme must be used as the sending application could modify the packet data after passing it for transmission. The zero-copy API is enabled by separation of network headers and payloads using scatter/gather transfers.

Checksum-offloading is also supported to reduce the per-byte overheads. This is made more difficult by the positioning of the TCP/IP checksums\footnote{As the checksums are stored at the beginning of the packet, the entire packet must be read by the NIC before the checksum can be written and the packet transmitted onto the network} and the possibility of multiple DMAs being used to transfer a single packet.

Trapeze largely focuses on improving the performance of very large transfers. For example, it supports MTUs of 64KB or larger. As a result facilities such as interrupt coalescing or suppression are not used: they have little impact when dealing with such massive packets.

Due to the reduction in data movement overhead Trapeze TCP/IP is able to demonstrate very high bandwidth (approaching the line speed of the underlying network, although the benchmark used does not touch the data and so does not represent a realistic test) with relatively low CPU usage for packets larger than 8KB.\footnote{It is at this point that the zero-copy optimisations can occur due to the restrictions on page alignment and sizes.} UDP latency is measured and shown to be less than that which was achievable at the time using Gigabit Ethernet.

2.6.5 Arsenic

Arsenic\footnote{Arsenic [72] is a Gigabit Ethernet-based network developed at the University of Cambridge Computer Laboratory. It uses a NIC (the Alteon Acenic) with two MIPS processors to perform some of the operations normally carried out by the operating system and so allowing a user-level application to access it directly. Applications are able to upload packet filters to describe packets which should be demultiplexed to that application and descriptor queues are used to transfer} is a Gigabit Ethernet-based network developed at the University of Cambridge Computer Laboratory. It uses a NIC (the Alteon Acenic) with two MIPS processors to perform some of the operations normally carried out by the operating system and so allowing a user-level application to access it directly. Applications are able to upload packet filters to describe packets which should be demultiplexed to that application and descriptor queues are used to transfer
buffers between the application and the NIC. Due to the increased complexity of
the NIC it is able to use per-flow strategies to offer better quality of service and
fairness between competing flows. It can also reduce the number of interrupts that
are raised.

The main focus of the Arsenic project was the production of a user-level TCP.
This TCP is able to deliver increased performance, both in terms of bandwidth and
application-to-application latency (when compared to the same hardware used in
a traditional kernel environment). For example, when dealing with TCP flows it is
able to achieve 8% higher bandwidth, using 9% fewer CPU cycles (or 34% fewer
CPU cycles if a zero-copy API is used) and requires less receiver side buffering (256 KB vs 400 KB per connection). This performance is attributed to the
increased efficiency of user-level networking. Latency can be less than half the
kernel equivalent, although they have the same performance if the applications
block for data rather than poll. In the user-level case latency is not increased to
the same extent by competing high bandwidth flows.

2.6.6 Other Cluster Technologies

While not strictly user-level TCPs, there have been a number of projects aimed
at reducing the overhead of TCP processing in server cluster and as such are of
interest here.

The Sockets Direct Protocol [69] (SDP, which is now part of the Infiniband
specification) and the Direct Access Transport [24] (DAT), are new protocols de-
dsigned to be able to use features of RDMA networks like Infiniband and VIA to
improve performance, while retaining (in the case of SDP) a BSD-sockets-like
interface to the application.

Intel have used a second processor in an SMP server to prototype a packet pro-
cessing engine. This “Embedded Transport Acceleration” (ETA) technique [77]
has a number of advantages, mainly stemming from the benefits of doing software-
level partitioning. For example, because the network I/O is all performed by a
single dedicated processor, it does not need to use interrupts and instead can poll
the network interfaces without worrying about the impact on other applications.
Scheduling also becomes simpler and so more efficient. The use of hyperthreads
2.7. Timers

Timers form an essential part of TCP. TCP uses two clocks (the fast timer ticks every 200 ms and the slow timer ticks every 500 ms) to measure time durations and schedule a number of different events. The fast timer is responsible for ensuring that any acknowledgements that have been delayed are sent. The slow timer both measures the round trip time of packets (this is important for calculating how long to wait before retransmitting lost data) and detects anomalies on connections by

on the dedicated processor can lead to further advantages such as parallelism in the packet processing engine to mask memory latencies. This combines to give impressive performance: throughput equals or exceeds that of a standard SMP Linux server (using both processors) and CPU utilisation is considerably less (80% of one CPU idle compared to all CPUs 100% utilised for the standard server).

A number of recent network interface cards provide support for performing TCP processing on the card itself, using a TCP Offload Engine (TOE), although performance has not always been as good as expected [2] and the architecture used must be given careful consideration [61]. Some have even provided additional support for technologies such as iSCSI, allowing disks to transfer directly onto the network.\(^{20}\)

Others (such as the Communication Services Platform [87], MemNet [75] and TCP Servers [74]) have proposed taking this a step further and dedicating a node within the network to TCP protocol processing and using a lightweight protocol to communicate with it. This has included commercial “TCP Termination Architectures” such as one from Voltaire [104] that terminate TCP connections at a router and then use SDP over Infiniband to communicate within the cluster.

Discussion of how to avoid the overhead of TCP network access is not limited to user-level TCPs. Work continues to study how this overhead can be avoided in traditional architectures [18].

---

\(^{20}\) Rangarajan et al mention a number of commercially available network cards that can do this in their work on TCP Servers [74].
2.7. Timers

scheduling events to occur after a period of inactivity.\textsuperscript{21}

This approach of using timeouts to detect failures has been highlighted \cite{111} for some time as sub-optimal. In this paper Zhang suggests that a decision based on a timeout is a guess and that for high performance external events should trigger failure recovery with timeouts used as a second line of defence. He highlights how it is particularly difficult to choose the amount of time to wait before retransmitting an unacknowledged packet, as this must be based on an estimate of the round trip time. This raises another problem of how to accurately measure the round trip time. Zhang also points out that external events convey information about what has gone wrong (and why), so allowing a more informed response. Finally, to have some confidence that a failure really has occurred, timeouts must be set conservatively (to avoid false detections of errors). This can often lead to unnecessary delays, particularly where the timeouts are overly conservative. This is increasingly an issue in modern, faster, networks where the times being measured are considerably less than their equivalents when TCP was designed.

As a consequence of the way timers are used in TCP, timeouts are scheduled and cancelled much more frequently than they occur and indeed more frequently than the ticks of the fast and slow timers. For example, a connection that is receiving packets of the maximum segment size will either schedule or cancel a delayed acknowledgement every time a packet is received. A delayed acknowledgement is only sent (at most) each time the fast timer ticks. The original implementations of TCP therefore made setting and clearing timers a fast operation, at the cost of each clock tick which was an $O(n)$ operation (where $n$ is the number of connections). Approaches based on Timing Wheels \cite{102} have been proposed as a solution to this. Timing Wheels (in their simplest form) represent an ordered circular list of events, with each element in the list containing the events that should occur at a particular time. On each tick of the clock, the next element in the list is examined and any events present are processed. This scheme is $O(1)$ (with respect to the number of timers) for setting timers, clearing timers and performing each clock tick, but does require that all timers are set for periods less than a fixed upper bound (to prevent wrapping around the circular list). An implementation of

\textsuperscript{21}The time and type of inactivity depend on the anomaly being detected. This is discussed in more detail in Chapter 6.
2.7. Timers

these algorithms for the BSD Unix version of TCP has demonstrated their scalability [27]. The Timing Wheel scheme has been extended [15, 29] to remove the need for the fixed upper bound and dynamically size the “wheel”.

As mentioned above the TCP slow timer is used both for scheduling timeouts and for measuring round trip times. Aron and Druschel criticise the use of this relatively coarse grained (500 ms) clock for measurements [4]. This leads to a high variance in the measured RTT, especially when the RTT is comparable to the clock period, which in turn leads to a very high estimation of the retransmission timeout (RTO). They propose separating the roles of scheduling and measurement and using a more accurate clock (1ms) for measurement. Scheduling of events is still done using the slow timer, and the retransmission timeout is still restricted to be no less than 2 ticks (0.5-1sec) to ensure correct behaviour. In related work [13] on TCP Vegas, Brakmo, O’Malley and Peterson demonstrate how the algorithm for calculating the RTO often results in estimates as high as 5 ticks (a delay of 2-2.5 secs) and never less than 3 ticks (1-1.5 secs) for a round-trip time of 100ms. They propose a more aggressive and responsive method for performing retransmissions: retransmitting once a RTT has passed, even if the timeout has not yet expired. To do this they record the (much more accurate) system time whenever a packet is sent and compare this to the system time when certain acknowledgements are received. This is very much in line with the approach that Zhang expounds, as described above. Both of these papers show how improved time measurement can also lead to improvements in the slow start phase of TCP.

TCP implementations originally performed a linear search of all the current connections on each timer tick to check if anything needs to be done. Aron and Druschel show how a simple change can lead to dramatic increases in performance [5]. They sort the list of connections so that all those in the TIME WAIT state are at the end of the list and that those in the TIME WAIT state are in the order in which they entered the TIME WAIT state. They can then terminate the loop that checks the timeouts of the connections as soon as one of the TIME WAIT state connections is encountered.\footnote{In the case of the slow timer they must also check the first few connections in the TIME WAIT state in case those connections have expired and should now be closed.} As the TIME WAIT state can be long compared to the amount of time that a connection is active (particularly for short lived
connections such as might be seen in a web server) there can be many more connections in the TIME WAIT state than any other. This change has a large impact on the time spent scanning the lists. In the test case used for comparison they were able to reduce the maximum CPU overhead of the timers from 25% to 0.36%.

Aron and Druschel extend their findings above to create “Soft Timers” [3]: a facility whereby events can be efficientlyscheduled with a granularity of tens of microseconds. This is achieved by executing the event handler at certain entry points into the kernel. These are accessed frequently in the normal course of program execution in response to operating system activity (system calls, page faults, interrupts, etc). The overhead of entering the kernel has already been made to perform the required operation and if an event handler is also executed at this time the cost of the context switch is amortised over them both. The times at which the trigger events will be called is neither regular nor predictable, so this technique can only schedule events probabilistically. However, experiments show that these triggers happen sufficiently frequently for it to be suitable for supporting a number of network operations. The authors suggest that rate-based clocking and polling of the network could both be implemented with low CPU overhead using this technique.

2.7.1 Discussion

TCP was designed for a very different environment to the one in which it is often now used. On one hand the network was slower, so the time to write data onto a network was orders of magnitude more than for today’s high speed networks. Round trip times were therefore higher than they are now and so any error in the measurement of round trip times represented a smaller percentage. On the other hand, operating systems were less advanced and the time measuring and scheduling facilities universally available were basic. To ensure portability TCP could not require that the system should be able to measure time accurately.

The pressures of reliable distributed systems, which benefit from accurate time measurement, have since resulted in support for low-overhead (particularly where relative rather than absolute time is needed) and accurate time measurement being present in most modern operating systems. It is therefore valid to question whether
the design of TCP’s timers, which was shaped by the available support at the time, is still optimal.

Recent developments in the Linux kernel have resulted in changes in this direction. Instead of using just two relatively coarse-grained OS timers as a clock to schedule TCP’s internal events, Linux has chosen to use the much more fine-grained general purpose timers available in the operating system directly. As a result it uses a number of different OS timers for each connection. These are maintained using a multi-level hash table (similar to the structure described in the Timing Wheels [102] paper mentioned above) and the algorithms are for the most part independent of the number of timers, resulting in good scalability as the number of connections increases [68]. However, the complexity of setting and clearing timers is increased (as it now requires linked list manipulation) and whether this trade-off is optimal has not yet been formally investigated.

Much of the research described above has been concerned with timers for a single kernel-level TCP. A system that could have a number of TCPs running concurrently at user-level has different requirements and facilities available to it. The Soft Timers approach (discussed in Section 2.7) for example achieves gains in performance by amortising the overhead of entering the kernel to check a timer. However, if the timers are implemented at user level, it will not be possible or appropriate to combine them with kernel access: this overhead will not be present and so their approach will not deliver the same gains.

\footnote{For example, the approach taken in recent Linux kernels requires access to the “jiffies” counter (a monotonically increasing variable) which is not accessible at user level.}
Section 2.6 has shown that user-level networking is a promising technology for cluster-based networking. This chapter considers how a TCP/IP stack can receive the most benefit from this style of networking. This motivates the creation of a prototype stack, with the ultimate aim (described in Chapters 4 and 5) of demonstrating how it can be integrated with other network components to provide a more efficient interface to the Internet.

3.1 Kernel to User-Level

Traditional kernel protocol stacks are executed in a different context to the application they are serving. The large overhead\(^1\) associated with context switching is one of the primary factors that has motivated the trend toward user-level networking. However (as discussed in Section 2.6) initial approaches to developing user-level TCP/IP stacks have used a similar structure to their kernel ancestors [14].

As many user-level TCP/IP stacks are direct ports of kernel stacks [72] they are structured to mirror the traditional setup in order to ease implementation. As

\(^1\)The magnitude of this overhead will depend on the size of the process, but is typically of the order of tens or hundreds of microseconds [59].
3.1. Kernel to User-Level

a result, those operations normally performed in the kernel, such as protocol processing, are instead performed in dedicated processes or threads. This has a number of disadvantages [22]. Firstly, although you have exchanged context switches for thread switches these are both considerably more expensive operations than a function call within a single thread. Secondly, protocol processing is done at some undetermined time after an application issues a request to send or receive data (at the mercy of the scheduler) and this can lead to artificially increased latency. TCP’s window size is sensitive to latency, so acknowledging data in a timely manner will inflate the window and increase the throughput.

Keeping the protocols in a separate thread from the application does have some advantages, particularly in terms of implementation. Although not explicit in the specifications, many implementations (and users) of the existing protocols and APIs expect the components of the network stack to be only loosely coupled through the use of queues. Many of the algorithms in use take advantage of these queues (which naturally occur at the boundary between contexts) to for example change the “chunk size” in which data are processed. Removing these different contexts and replacing them with a single flow of execution breaks many of these assumptions and this can result in unexpected behaviour. This is illustrated by Figure 3.1, where one thread writes to a queue with a chunk size of 500 bytes, while another is reading from the queue with a chunk size of 1024 bytes. Without the two threads and the queue, performing the change in chunk size is more complex (particularly if one chunk is not a multiple of the other). While it may be valid to return less than the requested 1024 bytes to the reader, the application may have come to expect it and so either break or perform suboptimally.

This chapter asserts that despite the challenges in implementation, there are performance benefits to be realised from a user-level TCP that does not have any thread boundaries. A user-level stack (called Cambridge User-Level TCP, or CULT) has been created to support this assertion. To highlight the novel aspects of CULT, the following section first outlines how other TCP stacks are structured.
3.2 Structure of TCP/IP Stacks

As outlined in Section 2.6, existing user-level TCPs have been structured in a variety of ways. The starting point for all of these is the kernel TCP. This is illustrated in Figure 3.2.\(^2\)

All of the network code resides in the kernel, along with an interface to the hardware (the device driver). Data are buffered at both of the kernel boundaries: i.e. between (i) the kernel and the application; and (ii) the kernel and the NIC. This has the following weaknesses:

- Interrupts are used to notify the kernel about the presence of received data. Interrupts are an expensive per packet operation, resulting in high overhead at high packet rates.
- The data are copied when they cross the kernel/application boundary (in either direction). This requires the involvement of the CPU and so reduces efficiency.
- The protocol processing is executed in a different context to the application that requested it. This reduces the accountability of this (potentially time consuming) operation and could result in unfair scheduling.

\(^2\)The series of illustrations that follows are simplified to highlight the structural differences. Many details have been omitted for clarity.
3.2. Structure of TCP/IP Stacks

Many user-level stacks attempt to address these points, as illustrated in Figure 3.3.

This approach has moved the network protocol processing to user level, where it is either executed in a dedicated protocol server process, or as a separate application thread. These will both require either a context switch or a thread switch on the data path. The different TCPs vary in the amount of the stack that is left in the kernel. The most typical case is that the demultiplex of incoming packets remains in-kernel, and this is shown by the packet filter in Figure 3.3. As a result there is still the need for a kernel context switch and a data copy at the kernel/user-level boundary.

More recently, networks such as Arsenic [72] have managed to remove the need for the kernel to be involved on the data path (at least in the case where blocking is not required) by placing the packet filter demultiplex on the NIC. This is illustrated in Figure 3.4.

This requires more advanced hardware. The protocol processing is still exe-
3.2. Structure of TCP/IP Stacks

![Diagram of network stack](image)

Figure 3.3: Structure of a typical user-level network stack

cuted in a separate thread to the application thread that has requested it and data are copied in both directions on the data path as part of the sockets API.

To counter the drawbacks of these approaches the CULT stack has been created with an alternative structure. This is described in depth in Section 3.3.

### 3.2.1 Protocol Issues

Before describing how CULT approaches certain aspects of protocol implementation differently, it is first useful to give some background on some of these concepts. This section outlines the creation and closure of TCP connections, and parts of the sockets API.
3.2. Structure of TCP/IP Stacks

TCP Connection Setup

TCP specifies as part of the protocol a three-way handshake for establishing new connections:

1. The client sends a connection request (a SYN packet) to the port on the server it is interested in.

2. The server responds to the client by acknowledging the connection request with a SYN of its own (the SYN-ACK)

3. The client then acknowledges the server’s SYN-ACK with an ACK.

This is shown graphically in Figure 3.5.³

³This figure is adapted from Figure 18.3 in TCP/IP Illustrated [93].
3.2. Structure of TCP/IP Stacks

TCP Connection Tear Down

To create a TCP connection requires three packets to be exchanged. To close one takes four. This is because TCP connections are full-duplex (data can be sent in both directions) and each direction must be closed separately. This is shown graphically in Figure 3.6.\(^4\)

API functions

In addition to the standard read, write and connection handling routines, stacks also need to provide support for more complex operations such as \texttt{poll()}, \texttt{select()}, \texttt{fork()} and \texttt{exec()}.\(^4\)

\(^4\)This figure is adapted from Figure 18.4 in TCP/IP Illustrated [93].
3.3. CULT

Poll() and select() both allow an application to test the condition of a set of connections. This may be to see if any have data waiting to be read, whether there are some new connections to be serviced, or if there is space to write some more data. In the case of select(), an application may block (i.e. wait) until one of these conditions is true.

Fork() allows a process to branch, making two copies of itself that then execute as separate processes. Exec() is in some ways similar, but replace the current process with a new one, but preserving the state of any open files, network connections, and such like.

The chosen architecture will impact the implementation of these functions. For example, many user-level stacks are unable to support poll() or select() operations that involve both standard OS file descriptors and user-level socket file descriptors. It is not enough, as many others have done, to only consider the data transfer functionality of a network stack.

The way that CULT is structured and how it provides support for these functions is discussed in the following section.

3.3 CULT

To address the problems with existing stack architectures, CULT has been developed. Its architecture differs from traditional stacks in a number of ways, which are explained throughout the rest of this section.

3.3.1 Hardware Used

The CULT stack has been designed to work on hardware from both the CLAN project and EtherFabric from Level 5 Networks. These platforms both provide a high-bandwidth, low-latency cluster network and have suitable programming interfaces to allow this to be exploited by the network stack.

CLAN formed the initial research platform, but with the closure of AT&T Laboratories — Cambridge Ltd., resulted in a lack of support for this hardware. Some minor faults in the CLAN hardware, and the small number of NICs available, led to the use of EtherFabric from Level 5 Networks as an alternative.
EtherFabric is able to support a number of modes of operation. Initially, it provided a normal packet-based interface, but crucially this is accessible from user level. It now also supports a distributed shared memory interface.

This need to support a variety of hardware and interfaces has had the positive impact that many of techniques used by CULT can be applied to standard packet-based networks. This is highlighted in Section 3.3.4 and later in Section 5.3.2.

3.3.2 Structure of CULT

CULT’s structure is illustrated in Figure 3.7.

The major differences between this and the other stacks are as follows:

- The separate thread/process to perform protocol processing has been removed. The application thread that requests the processing is used instead. It is this that leads CULT to be referred to as a threadless stack, as there are
3.3. CULT

no thread boundaries on the data path. Separate threads are used to implement some of the other functionality; this is explained and justified in the rest of this section.

• The kernel is not involved on the data path, other than to raise interrupts when required.⁵

• On the transmit data path, the data are not copied before being sent, even when using the sockets API.

• The data are demultiplexed by the NIC on the basis of their physical network connection. This requires much simpler hardware than packet filters in the NIC and does not require the involvement of the kernel. The network stack performs the mapping between these connections and the application level sockets.

3.3.3 Protocol Layering in CULT

In CULT the code is, from a design perspective, split into a number of layers.

**Network Interface:** The code with knowledge of how to access the hardware and manage the network-level connections.

**Protocols:** The code that implements the IP-based (including TCP, UDP, ICMP, etc.) protocols. This code is, as might expect be expected, largely unaware that it is part of an unusual network stack; the only differences being how it interfaces to the other layers.

**Application Interface:** The implementation of the Posix.1g sockets API. This must convert the normal sockets calls into operations that drive the network stack in the appropriate way. As the structure of the stack and underlying network are unusual, this layer has considerable knowledge of the underlying system rather than being just a simple wrapper.

---

⁵This property is shared with Arsenic
Although when specifying the system it is useful to divide the code into rigid layers, the efficient implementation of the specification requires more flexibility. Because separate threads have not been dedicated to each layer the interface between them must be much richer than normal. This allows each layer to call into the others when required. If different threads were used for each layer, simple message passing and queues between each one could be used to prompt the relevant actions, but with only a single thread at our disposal, that thread must be able to invoke those actions directly.

In normal stacks all communication between the sockets layer and the network interface layer goes through the protocol layer. In the CULT implementation the sockets layer and the network interface layer share knowledge about the underlying system, while the protocol layer is not specialised in this way. If, as in the normal case, all communication went through the protocol layer, it would become a barrier between the other two, forcing them to reconstruct information that is already available in the other. Instead the protocol layer has been moved to one side, allowing the sockets layer and the network interface layer to communicate directly. The protocol layer is now used as a tool by the other layers, rather than a means to communicate with the rest of the stack.

Figure 3.8: Flow of Execution through CULT code
3.3. CULT

Figure 3.8 shows how the path of execution flows between these different layers for a `send()` and `recv()` call.

The `send()` is the simpler of the two: the sockets layer passes the data to the protocol layer, which will attempt to send it on to the network interface layer. If there is no buffer or window space, the protocol layer will attempt to create some (“1” in the diagram) by (for example) processing any received acknowledgements. Once there is space, it is then free to pass the data to the network layer for transmission and return to the application.

The `recv()` path is more complex. When `recv()` is called, it immediately drops into the network interface layer (as this is where the queue of received packets is stored) and checks to see if there are any packets pending. If so, these are passed to the protocol layer for processing (“1” in the diagram). This may loop until one of the packets yields valid data for the application. The payload is then returned to the socket layer, which copies it to the location requested by the application (“2”) and signals to the network interface layer that it is now safe to free the memory that was being used by that packet in the queue of received packets (“3”) before returning to the application.

3.3.4 Threadless Protocol Stack

The decision not to dedicate a thread to the protocol stack presented a number of difficulties during implementation. Because the TCP/IP stack uses the same process as the application, if the application terminates then so does the stack and any pending data for transmission are lost. Sharing sockets between processes can be problematic, as can the efficient scheduling of protocol processing. The rest of this section details some of these challenges and how they are addressed.

**Blocking and Non-Blocking Semantics**

An application can typically access the network stack with either blocking or non-blocking calls. The blocking calls will wait until it is possible to perform the requested operation before doing so and returning. The non-blocking calls will return immediately with an error code if it is not possible to perform the requested operation straight away. While this distinction seems clear, there is some un-
certainty about how to define “the (in)ability to perform the requested operation straight away”.

For example, there are two queues that may contain received data (Figure 3.9). The first\(^6\) will hold unprocessed packets (of all kinds, some with payloads, some without) that have recently been delivered by the network. The second will hold the verified payloads extracted from packets as they are processed by the protocol stack.

\[\text{Figure 3.9: Received Data Queues}\]

There are two queues that contain received data. A dedicated thread moves data from one to the other, so blocking operations need only check the latter.

If a thread were dedicated to protocol processing (as in other systems) its roles would include taking packets from the first queue, processing them and placing the payloads on the second queue to await delivery to the application. When the application calls \texttt{recv()} and does not wish to block, it is sufficient to check the second queue: if it is empty the operation is judged to block and so an error is returned. The presence of packets on the first queue is irrelevant as these will get processed by the dedicated thread the next time it is run. In contrast, the threadless CULT system must examine both queues. If there are no packets on the second queue, but packets on the first then it is possible that the operation could succeed immediately \textit{if the packets on the first queue were processed}. However,

\(^6\)Strictly speaking there could be many of these, as there can be a many-to-one mapping between physical network connections and sockets. However, for the purposes of this discussion it is only necessary to consider the simple case.
presence of packets on the first does not guarantee that data will become available for reading if the packets are processed: they could be acknowledgements, or be corrupted and so dropped for example. These packets must be processed before it is possible to make a decision about whether or not the operation will block. It isn’t possible to take the other approach (i.e. return if the second queue is empty) as it will remain empty until the packets on the first queue are processed. In the absence of a dedicated thread to do this, the second queue would remain empty indefinitely and the system would grind to a halt.

The initial reaction to the suggestion of performing the protocol processing at this point is often that a non-blocking operation should return quickly if there is nothing to do and by delaying the return (albeit with useful work) it will be detrimental to the performance of the application. This can be countered as follows: for the system to proceed the packets in the first queue must be processed. This can either be done by a dedicated thread, or by the application thread when a \texttt{recv()} operation is requested (i.e. lazily, as described above). If the former is chosen, the application will have to be pre-empted in order to allow the protocol thread to run.\footnote{This is not the case where the $T = N - 1$ (T = number of application threads, N = number of CPUs) as the single spare CPU could be used by the protocol thread at no cost.} In both cases the same amount of work will have to be done (and so the same amount of time taken away from the application) but if a different thread is used there is the additional overhead of the thread switch. In the lazy case, the application may also benefit from cache effects as the data will have been processed just before it uses them and so they are likely to already be present in the cache. Therefore, doing the processing at the time the call is made may actually increase the performance of the application rather than hinder it.

Another possible criticism of our approach is the need to perform timely acknowledgements. The window algorithms in TCP rely on the receiver acknowledging data that have arrived in order to allow more data to be transmitted. To maintain a streaming traffic pattern (as opposed to a bursty one) the acknowledgements need to be returned as soon as possible. If protocol processing is only performed when the application calls \texttt{recv()} there could be some delay between the data being delivered and the data being processed (and so acknowledged). This could reduce the performance of the network.
3.3. CULT

<table>
<thead>
<tr>
<th>Read Size</th>
<th>Time Between Reads</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (&lt; window)</td>
<td>Smooth</td>
<td></td>
</tr>
<tr>
<td>Large (&gt; window &lt; buffer)</td>
<td>Smooth</td>
<td>Bursty for CULT</td>
</tr>
<tr>
<td>Very Large (&gt; buffer)</td>
<td>Smooth</td>
<td>Bursty for both</td>
</tr>
</tbody>
</table>

Table 3.1: Impact of read frequency and size on performance

If there is no dedicated thread to perform protocol processing and the application does not call `recv()`, the sender will stop transmitting data once the window is exhausted. If there is a dedicated thread to perform protocol processing, the thread will continue to acknowledge incoming data, even if the application is not calling `recv()`. The stack is however forced to buffer the data until it does. There is a limit to the amount of buffering that is permitted and once this limit is reached the stack will close the window to prevent the sender transmitting more data. The end result is the same in both cases.

Table 3.1 shows that the only difference in performance will come if the application reads infrequently and those reads are for large amounts. In this case, the buffering by the dedicated thread would absorb this burstiness and the underlying network can still stream, but in the CULT case the infrequent reads could result in bursty traffic. If it reads very large amounts infrequently, the buffer size will not be sufficient to smooth the burstiness. If it only reads small amounts infrequently, the application itself is the bottleneck and so the network stack performance is not important.

Connections

The concept of a connection\(^8\) exists at many different levels in the stack:

---

\(^8\)The term “connection” is poorly defined in the context of networks. In some cases it can mean the existence of a signalling channel that must be used before and after data can be sent (as in connection-oriented or connectionless network). In this context it is used it to mean the
3.3. *CULT*

- At the network layer, because CLAN (and other networks used — such as EtherFabric — are able to emulate this) is a connection-oriented network, there are the *network connections*.

- At the protocol layer, there are the *protocol control blocks* (PCBs) for each transport protocol connection\(^9\).

- At the sockets layer, each *socket* will represent a channel of communication available to the application.

Even in the absence of a connection-oriented network layer CULT is still able to emulate network layer connections (and so realise some of the benefits, such as an easy packet demultiplex) using the protocol described in Section 5.3.2. The out-of-band messaging channel discussed there is not only used for load balancing, but also for connection setup (i.e. creation of the distributed message queues) on EtherFabric, which does not provide a built-in means to do this.

In general there is a one-to-one mapping between sockets and PCBs (it is shown below how, during TCP connection setup, this is temporarily not the case). To allow packets to be correctly demultiplexed there is a many-to-many mapping between network level connections and PCBs. In many cases this will be one-to-one, but there are examples where a set of network level connections will be associated with a set of PCBs (e.g. UDP broadcast).

Even in the common case of a one-to-one mapping, the mapping can change over time. For example, during TCP connection setup (discussed in detail below), or when *sendto()* or *recvfrom()* socket calls are used.

Maintaining this mapping between sockets and network connections illustrates the need for these two layers to communicate directly; going via the connectionless IP layer would require complex reconstruction of information above and below it that was already available in the other layers.

---

\(^9\)The concept of a connection as defined above exists even for datagram/connectionless protocols such as UDP
3.3. CULT

TCP Connection Setup

In the case of CULT, an extra step precedes the standard exchange described in Section 3.2.1. Before the initial SYN can be sent a network-level connection must be established.

Supporting a connectionless protocol such as IP over a connection-oriented network and the associated signalling problems are well studied topics. Classical IP and ARP over ATM is an obvious example [53]. As a result, CULT takes a functional (rather than research) approach to this problem.

The implementation in CULT involves its own (two-way) handshake for creating the network-layer connection. Because the protocol layer has no dedicated thread and providing callbacks and checks to ensure that the application thread was diverted to establish network connections at the right moment would be complex, a separate thread is used solely for this purpose, but this should not impact the fast-path for data. It communicates the new connections to the rest of the stack using an asynchronous message queue.

The asynchronous message queue is structured so that there is a single writer (the listening thread) but multiple readers (every PCB that is interested in new connections, such as one in the TCP LISTEN state). All the readers check this asynchronous message queue when they attempt to do protocol processing and can choose to block on it if they wish to wait indefinitely for a connection. Not all readers of the queue will be interested in all incoming connections (they may only be listening on a single port for instance). The handshake for establishing the network connection includes enough information (source and destination IP addresses and, if applicable, ports) to allow them to make this decision in advance of any IP segments being delivered.

As well as a new network-level connection, a new PCB and a new socket must also be created. The way this is performed and how the mapping between the components changes at each step, is illustrated in Figure 3.10.

The network connection is established (Figure 3.10(b)) and a listening PCB is associated with it (Figure 3.10(c)) as described above. It is important that an association remains between the network connection and a socket (via a PCB) at all times: all activity is driven from the layer above (by the application) and
3.3. CULT

(a) Application calls `accept()`  
(b) New network connection established  
(c) Network connection mapped to listening PCB  
(d) SYN delivered and processed, SYN-ACK sent  
(e) Sockets layer notified of new PCB  
(f) ACK delivered, new socket created.

Figure 3.10: TCP handshake processing.

As the handshake progresses the data structures associated with the connections at each layer are built up from the bottom. As each one is added the mappings are changed to point to a structure in layer above.
so without an association it will not get processed. This is demonstrated by the series of diagrams in Figure 3.10 - as each component is added, the mappings are changed to maintain a vertical link from bottom to top.

The first SYN is delivered. The stack creates a PCB for the new connection (Figure 3.10(d)) and notifies the sockets layer that it has had a connection request (Figure 3.10(e)). The application thread that is responsible for the listening PCB (the one that has called accept()) continues to drive the protocol processing of the three-way handshake until the third packet is received (i.e. the TCP connection is fully established) at which point the accept() call can create a new socket (Figure 3.10(f)) and return.

Although each socket is usually associated with a single PCB it is clear from the diagram at step (e) that the listening socket has two PCBs. This is to ensure that the third packet of the TCP handshake is processed; it is not until this packet is received that the new socket is created, therefore the listening socket must take responsibility for it in the mean time.

There can be significant delay during the transition from (a) to (b), (c) to (d), and (e) to (f) in Figure 3.10 as these all involve communication across the network with the connecting client. This does not present a problem for the threadless CULT stack when accept() is used, as it is reasonable to either block at these points (if a blocking call is made) or return and wait for the next time accept is called (if a non-blocking call is made). However, connections can also be established using poll() and select() and these calls will typically attempt to service multiple sockets at once. These calls cannot block waiting for the connection to be established, as the other sockets would be neglected until this happens (which could be a long time). The poll() and select() calls must therefore always be able to advance the connection process to the next step when needed, regardless of where in the process they first meet it, without blocking on any one socket. This is where the complexity of the above connection mapping and migration becomes necessary.

Much of the extra complexity described in this section is present to implement the TCP protocol, rather than the simpler UDP or other transport protocols. Unfortunately, as well as requiring more complex connection setup, TCP also has complex requirements for closing connections.
3.3. CULT

TCP Connection Tear Down

As with connection setup, connection tear down requires some additional complexity over the standard exchange described in Section 3.2.1. The problem for a threadless user-level TCP stack such as CULT is twofold:

- The application will often quit after calling \texttt{close()}. A user-level stack that is linked with the application will therefore also exit. If the connection termination is not complete, or some other action (such as retransmission of an earlier packet) is required, the stack will no longer exist to deal with this.

- If there are data waiting to be sent when \texttt{close()} is called, these must be dispatched (and the acknowledgements received as a result processed) using the application thread. The application will not service the closed socket again once \texttt{close()} returns, therefore there will be no thread available to continue the processing.

For user-level stacks such that retain a persistent process either in the kernel or as a server [57, 99], this can be solved by migrating control of the closed connection to the kernel or server portion of the stack. For CULT, the solution to both these issues is to force the \texttt{close()} call to block until these conditions are met. (i.e. The four packet tear down completes.)

It should be noted that synchronised termination of the application and the stack does have some advantages. For example, it acts as a crude form of garbage collection: no network buffers can be left hanging around, as they might with a scheme such as “I/O deferred page deallocation” [16] (part of the Genie I/O framework).

Timers

TCP requires that two functions are called regularly. The fast timer must be called every 200 ms, the slow timer every 500 ms. In the CULT architecture, no guarantees can be made about when, how often, or for how long the application will make calls into the protocol stack. It is only during these calls that the thread can be used for protocol-related activity, therefore to perform regular actions requires that the timers have a separate thread.
3.3. CULT

This can be justified (in a way that the dedicated thread for protocol processing could not) because the timer functions do not form part of the data path, therefore dedicating a thread to this function does not lead to any thread or context switches while processing packets. The implementation of timers in the CULT stack is discussed in detail in Chapter 6, including how this thread avoids “active” connections (ones that are being serviced by the application) and so does not interfere with data transfer operations.

Fork

Although not affected by our decision to use the application thread to drive the protocol stack, the issue of how to deal with fork() is one that affects us because the protocol stack is linked with the application. Fork() spawns a new process (the child), initially with the same state as the calling process (the parent). As a result, both processes have access to the file descriptors and sockets that were created before fork() was called and could both go on to use them. For a kernel stack where the state for the sockets is not kept in the address space of the process this does not represent a problem, as there is only one copy of the state. For our user-level stack, the state is kept in the same address space as the application; therefore when fork() creates the new process, the state is also copied. If both parent and child went on to use the sockets, the two copies of the state would quickly diverge, whereas the kernel stack state would be consistent (as there is only one copy of it).

In general, most applications that use fork() ensure that the subsets of sockets used by the parent and the child form a partition (i.e. each socket is used by only one process or the other). Each process calls close() on the set of sockets that it will not use. CULT has therefore been restricted to supporting the ability for multiple processes to close the same socket. If this case was not dealt with, when one process tried to close the sockets that it did not plan to use, the connections at the protocol layer would also be closed, preventing the other process (which did plan to use that socket) from making use of it.

To do this an an 8-bit reference count is kept for each socket in a region of memory that is shared by both parent and child (and any others that have been
forked). When a socket is created the reference count is initialised to 1. Whenever \texttt{fork}() is called, the reference count of all active user-level sockets is incremented by 1, to reflect the fact that there are now two (or more, if \texttt{fork}() has been called before) instances of each socket. Whenever \texttt{close}() is called, the reference count is decremented by one and the state for that socket (other than the reference count) is removed from that process. If the reference count is now zero (ie. there are no more processes using the socket) then the protocol connection is also closed. This ensures that each process can call \texttt{close}() on the socket without affecting the other processes ability to use it, but that any state for the socket is cleared up appropriately.

**Exec**

As with \texttt{fork()}, \texttt{exec()} is another function that requires some consideration despite not being directly related to the functionality of a network stack. \texttt{Exec()} replaces the current process image with a new process image. Crucially, any open file descriptors remain open and are usable by the new process image. This clearly represents a challenge for a network stack that is in the same process as the application: if the application process is replaced, so is the network stack and all knowledge of the file descriptors.

To implement \texttt{exec()} for CULT would therefore require kernel support. The state of the open file descriptors would need to be preserved over the switch, requiring a hook to save the state before replacement and another to prime the new process with the same state before execution recommenced. Such techniques have been well researched in relation to progress migration, for example as part of the Sprite [31] operating system and the Zap [66] project.

Although straightforward in concept and clearly possible to do, it would be difficult in practice and would require more implementation time than is available during a PhD. As the topic has already been well researched and the function was not required for any of the tests and applications used, the implementation of \texttt{exec()} in CULT has been left as further work.
Event Distribution

The network driver provides a single event queue that is used to communicate with the user-level process. Providing access to this queue that is compatible with the structure of the CULT stack is complicated by a number of factors:

- This queue contains events relating to both transmission and reception of packets and also for all the different active connections.

- Each application thread may only be interested in one type of event, and perhaps just one connection and may not wish to process events of other types. Alternatively it may want to wait on many connections or types, to provide `select()`-type functionality.

- Finally, the use of a separate thread to service the event queue was rejected and so it must be accessible by either a single thread or many if concurrent access occurs from different application threads.

To achieve this, a novel approach to demultiplexing events is used. Traditionally, a producer-consumer model would be used, where events are demultiplexed by a single “I/O thread” (the producer), which then divides the work of processing the events among a pool of “worker threads” (the consumers). An alternative, termed the leader-follower model [86], removes the need for the I/O thread by having the worker threads take it in turns to access the event queue directly (this thread is then termed the “leader”), removing a single event and then promoting another (“follower”) thread to leader before processing it. This assumes that all the follower threads are equivalent and all equally able to process all events. This is not the case for CULT, where particular threads will only be interested in particular connections and event types. To address this, a combination of the leader-follower and producer-consumer schemes (termed “leader-consumer”) is used as follows.

When a thread wishes to wait for an event, if there is no leader it becomes the leader and services the event queue. If there is already a leader, the thread becomes a consumer. Events are sorted by the leader depending on the event type and placed in fixed length circular queues.
3.3. CULT

When a thread becomes a consumer it first checks the queues it is interested in and if an event of interest is present it takes it and processes it. If there are no pending events of interest to it, it blocks on a condition variable.

If the leader receives an event that is of interest to it, it services the event and promotes one of the consumers (if any) to leader (as it would in the leader-follower model). If it receives an event that is not of interest to it, it sorts it onto one of the circular queues and signals the appropriate consumers (if any) through the condition variable (as it would in the producer-consumer model).

This can be summarised as the following set of decisions made when a thread wishes to wait for an event:

1. Is there already an event of the correct type waiting? If yes, take it and leave, if not, wait for an event as follows.

2. Is there already a leader? If not, become the leader, if yes, become a consumer.

3. If it is now the leader, service the event queue:
   
   • For each event that arrives, sort it into separate queues according to its type. If there are any consumers, signal them that there is a new event of interest.
   
   • If an event is of the type the leader is waiting for, take it, signal any consumers that there is now no leader and leave.

4. If it is a consumer, wait for a signal from the leader.

   • When signalled, check the list of events of the type it is waiting for. If there is one of the correct type, take it and leave.
   
   • If a waiting consumer is signalled and there is no leader, become the leader and service the event queue as above.

This “leader-consumer” approach combines the advantages of the leader-follower model where no dedicated I/O thread is required, but maintains the flexibility of the producer-consumer model that allows different threads to service different
3.3. **CULT**

types of events. In addition, it works well with either one or many threads, but
does require a lock to control which thread becomes leader (which, in the absence
of any consumers, is wasteful).

**Poll and Select**

The `poll()` and `select()` socket calls allow an application to test the state of
a number of sockets in a single operation. Select will also allow the application to
wait for one of the sockets to become readable or writeable (or have an exceptional
condition).

A disadvantage of most user-level network stacks is the inability to use `poll()`
or `select()` on a mixture of user-level sockets and OS sockets or file descriptors. This severely limits the applications that can, in practice, use the user-level
network without modification.

The CULT stack does not have this limitation. To do this it makes use of
the network driver’s asynchronous event queues which were discussed earlier in
this section. Whenever there is an event of interest the network driver (or NIC,
depending on the hardware in use) places an event on the event queue. The driver
also provides an OS file descriptor for the event queue that can be tested in the
same way as any other file descriptor: it becomes readable when the queue is not
empty.

The following description describes how `select()` is implemented and al-
though much of the same applies to `poll()` it is significantly simpler as it never
has to block.

When `select()` is called, the application passes three sets of file descriptors.
The first (`readfds`) will be watched to see if a read operation will not
block, the second (`writefds`) will be watched to see if a write operation will
not block and the third (`exceptfds`) will be watched for exceptions.

The CULT wrapper for `select()` examines each set of file descriptors and
checks each one against the list of user-level sockets. If it finds a match it removes
that file descriptor from the set and adds the file descriptor for the corresponding
event queue to the `readfds` set.\(^\text{10}\) It also checks the event queue in question to see

\(^{10}\) The event queue file descriptor is added to the read set (rather than the set that the socket file
if there are any events already pending. If so, it flags it.

Having munged the file descriptor sets in this way to contain a combination of OS file descriptors and event-queue file descriptors, the subsequent action depends on a number of factors:

1. If there are no CULT sockets in the sets, the standard OS `select()` is called with the original arguments and the result returned without further processing.

2. If there are no native OS file descriptors in the sets and some of the CULT event queues have pending events, the sockets are examined to see if they meet any of the conditions that the application is interested in. If none of them do, it drops to the most general case (step 4) below.

3. If there is a mixture of CULT sockets and native OS file descriptors, and some of the CULT event queues have pending events, then to ensure fairness the OS `select()` is called with a zero timeout in case any of the OS file descriptors immediately meet their conditions. The event queues are also examined to determine which CULT sockets, if any, already meet their conditions. The results of these two are combined, and if there has been some activity of interest to the application it returns.

4. Otherwise, there is the most general case, where there is (potentially) a mixture of OS file descriptors and CULT sockets and none of them are immediately of interest. In this case the munged file descriptor sets are passed to the standard OS `select()`, with the original timeout, to block waiting for some to change status. As before, when `select()` returns, any event-queue file descriptor activity is processed to check if it is activity that the application is interested in and this result is combined with any OS file descriptors that have changed status.

This step loops until there is either a result to return to the application, or the timeout has expired.

dercriptor was in) as all types of events will make the event queue readable.
This series of steps is designed to ensure that the highly loaded case is optimised and that whenever possible it avoids calling the (relatively high overhead) OS `select()`. In a busy application using exclusively CULT sockets, it should rarely need to resort to making the system call. If a mixture of OS file descriptors and CULT sockets is used, then performance will be reduced, but this is a significant improvement on not supporting this case at all.
Chapter 3 considered how to structure a user-level TCP stack. This chapter extends that to show how the new structure can be used to efficiently provide the two most important operations: transmission and delivery of data. This leads to the use of the gateway to assist in TCP retransmissions.

4.1 Delivery

Section 3.2 outlined how a number of other TCPs demultiplex a stream of IP packets to individual sockets. With the exception of CULT, the demultiplex was performed either in the kernel, or by the hardware. CULT avoids this by taking advantage of the physical network and using connections at the network layer to identify which socket packets are destined for. A NIC can demultiplex these network connections much more easily than it can TCP/IP packets, but this approach does require a connection-oriented network to provide the network connections.
4.1. Delivery

4.1.1 Header and Payload Separation

The network connection is represented in CULT by a pair\(^1\) of Distributed Message Queues (DMQs). (DMQs are described in Section 2.5.8). One DMQ is used to transfer the headers (both IP and the transport layer header, along with any options), the other to transfer the payloads.

There are a number of reasons for separating payloads and headers in this way:

- The headers and payloads are processed at different times by different components. If they were stored sequentially in memory, the memory would quickly become fragmented as headers were freed after processing, but their corresponding payloads had not yet been delivered to the application.

- To avoid copying, the memory containing the payloads can be exposed to the application when data are delivered. If the payloads were separated by headers the application would have to be aware of this, rather than being able to read any amount it chooses sequentially.

- TCP limits the amount of unacknowledged payload that can exist through the TCP window. If the window has become closed the sending TCP should still be able to transfer packets that consist only of headers: an ACK for example. Separating the headers and payloads ensures that if the payload queue is full, the header queue can still be used as expected.

The disadvantages of this approach are:

- The transfer of data at the network layer becomes more complex, as the sender must write the headers and payloads separately. This in turn requires more complex synchronisation to ensure the receiver does not attempt to read a packet where only the header or payload has been delivered.\(^2\)

- Less efficient use of memory: The optimal relative size of the two DMQs will depend on the ratio of payload size to header size for the current flow. It

---

\(^1\)Each connection is half-duplex, so in fact for duplex operation four DMQs are required.

\(^2\)This is further complicated by the choice of PIO and DMA - PIO writes can overtake DMA writes if the DMA engine is busy. Typically headers will be written using PIO as they are small, and payloads using DMA as they are larger. Therefore the header and payload may not always arrive in the same order that they were written.
is not possible to dynamically size the queues so they must be large enough to ensure that one does not unduly limit the other. There sizes must also integrate well with buffers such as the TCP window.

4.1.2 Virtual Ring Buffer

If a pointer into the payload queue is to be passed to the application to allow it to read data directly without copying, the application needs to be able to read any amount sequentially from any point in the queue. Separating the headers is part of this, but the circular nature of the DMQ must also be hidden. To do this a Virtual Ring Buffer [41] (VRB) is used. This involves mapping the physical memory page that contains the DMQ into two adjacent virtual memory pages. The entire queue can then be read sequentially, starting from any point in the queue, as shown in Figure 4.1. This restricts the DMQ size to be a multiple of the page size and increases consumption of virtual memory.

![Figure 4.1: A Virtual Ring Buffer](image)

The shaded portion represents readable data. The application is able to read it all in a single operation as the buffer is mapped into a second page (the dotted region) adjacent to the first.

4.1.3 In-Place Processing and Delivery

The separation of payloads and headers facilitates the ability to perform protocol processing and delivery from the same buffer.

The standard DMQ has two pointers (a read pointer and a write pointer). To allow processing and delivery from the same DMQ, the read pointer is split into two
4.1. Delivery

- a processing pointer and a delivery pointer. The processing pointer will always be between the write pointer and the delivery pointer and so is not immediately of interest to the remote node.³

Section 2.3.3 discussed how the semantics of the sockets API require a copy (or some virtual memory system manipulation) on `recv()` to move the data into the buffer provided by the application. For a user-level stack, virtual memory system manipulation would be difficult (not to mention risky) and the constraints (such as the requirement for application buffers to be a multiple of the page size) of this approach make it unsuitable for CULT.

Performing a copy at this point is the simplest way to support existing applications without modification and this facility is provided to ensure backward compatibility. While a copy is a high-overhead operation, some of the cost of copying can be absorbed by performing it at the right time: if the application is about to access the data anyway, a copy immediately before it does will increase the probability that the data are already loaded into the cache.

Modifications to the sockets API allow us to return a pointer to a buffer instead of copying the data into an application buffer. This is subtly different to the existing work described in Section 2.3.3, as rather than resulting in moving the data to the application memory space, it grants temporary access to the buffers that remain in the stack memory space. This is arguably suitable for a (large) class of server applications which do not need to preserve the data they receive from the network. Instead, they process what are typically requests and then discard them. If the data did need to be preserved, a copy would be required and so there would be little benefit. This approach does require some modifications to the application. It also raises the question of when to free the data in the DMQ (i.e. increment the delivery pointer). Incrementing this pointer signals to the remote node that it is now able to use that region of the DMQ for the next packet. Therefore, if this pointer is incremented as the data are passed to the application, the data could get overwritten before the application has finished using it. The most robust approach is to make a second addition to the sockets API to allow the application to explicitly signal that it has finished with the region of memory.

³It is possible that exposing this information could be useful for other functionality such as load balancing in the same way that the read and write pointers are, as discussed in Chapter 5.
An alternative method (which is less robust, but requiring no application changes) assumes that the application processes the packets sequentially and does not perform subsequent reads until it has processed earlier ones. The memory passed to an application can then be released next time a \texttt{recv()} operation is called. This is unlikely to be compatible with situations where middleware separates the stack and application.

Finally, it is possible have an approach that leaves control of the buffers with the application, rather than with the network stack. When the application calls \texttt{recv()} and “pre-posts” a buffer for the data, this buffer can be published to the remote node and the payload placed directly into it. Headers would be placed into a DMQ as before.

**Out of Order Packets**

TCP packets that are delivered out of order require special treatment. As they are not immediately deliverable to the application and their presence in the DMQ could prevent further packets being transferred, they are copied out of the DMQ to await further processing. If they are subsequently able to be delivered they can be queued in the normal way. This extra overhead can be justified by its rarity and it does not impact the fast path for in-order packets.

### 4.2 Transmission

Section 4.1 described the techniques used to increase the efficiency of delivery. Zero-copy transmission has a different set of associated problems. Existing attempts at zero-copy TCP [19, 40, 97] have either modified the sockets API, or used copy-on-write page remapping.

The sockets interface (particularly when combined with TCP) was not designed to perform zero-copy operations:

1. When \texttt{send()} is called it returns once the data have been placed on TCP’s send queue.\footnote{While this is not explicitly mandated, all implementations that have examined do so.}
be done before the packet is transmitted and it could be some time before it is safe to discard the data. As a result the stack is required to copy the data to prevent it being overwritten by the application.

2. If this were not the case and the integrity of buffer could be guaranteed until the data are written, there would still be another problem. Many high performance network cards use DMA to transfer the data to the network and the DMA request may not happen immediately. The data must be preserved until the DMA has completed.

3. Even if this is achieved without copying, in the case of a TCP packet the stack still needs to preserve the data until a TCP acknowledgement is received. Until this time the stack may be required to retransmit the data. This will take at least one round trip time (RTT) and when communicating over the Internet RTTs can be large.

In the following sections a number of alternative schemes for addressing these problems are discussed.

4.2.1 Update the Sockets API

Firstly, because the problem stems from a weakness in the sockets interface the obvious choice would be to remedy this by changing the interface. It needs some way to notify the application when it is safe to re-use the buffer. Many zero-copy APIs have been developed [82, 101] to do just this, including the IEEE standard for Asynchronous I/O [45]. Many would argue in favour of this option; effort should not be expended to provide workarounds for outdated APIs. However, while that might be an ideal, in reality, if there is a solution that (all else being equal) can fix the problem without changing the API, it will be more popular. The resistance to changing APIs should not be underestimated.

4.2.2 Alterations to the Transport Protocols

It could also be argued that retransmits by servers are, in many cases, unnecessary. Were this burden removed from the server the need to wait for an ACK would go
away. Where the data being served are static, the client could simply re-request any erroneous or missing data, especially if Application Level Framing [20] is used. Where the content is dynamic the client would have to ensure that it received a coherent set. There may be cases where this is not feasible and so, in those at least, retransmits are desirable.

This could be realised by using UDP as the transport protocol in place of TCP and many modern servers and clients (for example RTP or NFS) effectively do just that. It is safe to assume that those servers that have chosen to use TCP have done so for a good reason and so this is not a suitable solution for them.

### 4.2.3 Blocking semantics

The application will (or at least can) re-use the buffer as soon as the `send()` returns. Therefore, if the buffer can not yet be re-used, the stack could simply not return from the `send()` call. Instead of blocking until the data have been queued on the TCP send queue, it could block until the data have been sent, or the DMA operation has completed, or even until the TCP acknowledgement has been received. However, blocking until a TCP acknowledgement is received for each `send()` in the worst case reduces TCP from a streaming protocol to a ping-pong protocol. Only one `send()` per socket can be in-flight at any one time.\(^5\) Because of the threadless nature of our TCP stack it is easy to experiment with this change in blocking behaviour. It was observed that moving from a system where it blocks until the DMA has completed to a system where it blocks until an acknowledgement is received resulted in at least half the throughput on a local area network.\(^6\) It is clear this is not suitable as a solution, but blocking the `send()` until the DMA has completed may be justifiable. This would solve the problem for transport protocols such as UDP where retransmits are not supported.

If this change to the blocking semantics is used, the stack would still need some way to guarantee access to the data for protocols which do perform retransmits. Instead of avoiding the copy, it may be to necessary to instead look at ways

\(^5\)This is not the same as a single packet being in-flight at once, as applications often pass large blocks of data in a single send operation.

\(^6\)The exact degradation will be strongly related to the ratio of bandwidth to latency of the network.
that the data could be copied with low overhead. Some suggestions are discussed in the following sections; these are all based around performing the copy at the same time as writing the data to the network and so absorbing some of the overhead of the copy into the network write. This is possible as the copy and the write are essentially very similar operations, especially in a distributed shared memory based network such as CLAN or EtherFabric.

4.2.4 Memory Multicast

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

Micro-ops

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

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

NIC Assisted Retransmission

A more feasible solution would be to add an expansion card to accompany the NIC. This would be relatively simple, consisting of an FPGA, some memory and a PCI bridge. This card would snoop the PCI bus, watching for writes to the network. It would store the data associated with such writes in its RAM. While not used for retransmission, a similar approach [92] has been suggested for related problems and the Afterburner [28] card is hardware rather like that described above. Should a retransmit be necessary the stack could then request the expansion card use DMA to transfer the relevant data to the NIC and in so doing perform the retransmit. This achieves preservation of the data with no additional CPU load and no additional bus bandwidth consumed, but it does have some disadvantages. For example, there is the expense of the extra hardware required. This could be reduced by integrating it into the NIC but it would still dramatically increase the hardware costs and limit scalability, especially for a simple card like the CLAN NIC.

4.2.5 Copy On Write

A different approach would be to only copy the data if necessary and avoid it in all other cases. It is only necessary to copy the data if the application tries to re-use one of its recently used buffers.\textsuperscript{7} It would be straightforward to implement a copy-

\textsuperscript{7}Many servers will be dealing with static content and so may never change the data in buffers it has passed for transmission.
on-write system where when a buffer was passed to the stack that page was marked as “in-use” until the corresponding TCP acknowledgement is received. If the application tries to write to that page in the meantime the page would be copied. Clearly this requires manipulation of the page table entries by the protocol stack. This would be easy if the stack were in the kernel (and in fact this has been used by kernel stacks [40]), but from user-level it would be necessary to either make a system call (which would negate much of the benefit of having the stack at user level) or perform user-level memory management and page table manipulation, which in turn would need selective TLB flushing. It would also only give benefit to those applications who normally would not require the buffers to be copied. While many may fall into this category, there are some notable exceptions (particularly benchmarks) which would not. In these cases, the performance would be worse than if you had just copied the data to start with because of the additional overhead in page table manipulations.

4.2.6 Discussion

The techniques outlined in this section for avoiding the copy required to support TCP’s retransmit algorithm all suffer from serious drawbacks. The following section proposes a novel alternative approach: making use of the implicit functionality of the network itself (in this case the Distributed Memory Queue) to perform operations required by higher layer protocols with little additional overhead. This avoids duplication at the cost of integrating network layer technology into the protocol implementation. In essence, this examines the optimal split of protocol processing between software, local hardware and network hardware.

4.3 Protocol Gateway and Retransmission

There has been significant amount of research and development recently in the field of TCP Offload Engines (TOE), leading to a number of commercial products. TOE involves using a NIC (or possibly a dedicated external machine [74]) to perform some portion of the TCP protocol processing, but the benefits that this brings are far from clear [89, 61] and often do not result in the applications
themselves seeing increased performance [84]. Our user-level TCP clearly takes a different approach by performing all network processing in software. However, to investigate the split of protocol processing between software, hardware and the network, this section explains how we use the cluster gateway to assist the nodes in TCP processing: it removes the need for the node to copy data for TCP retransmission.

4.3.1 Gateway Buffering for Retransmission

Rather than require the server node to preserve the packets it has transmitted until they are acknowledged, CULT provides a means by which the gateway can do this more efficiently. The gateway must access all data flowing out of the network and so can buffer it in memory at little extra cost to itself. As a result, a copy of the data is preserved by the cluster without any extra CPU, bus, NIC resources, or network bandwidth being used on the server nodes. This is illustrated in Figure 4.2.

Memory Costs

There is the additional cost (most notably memory) involved at the gateway, but this is less critical than the cost of the NICs as there are far fewer gateways than NICs. Also, by having the retransmit memory shared by all the nodes in the cluster it is possible to make more efficient use of this resource. Secondly, the gateway will be using a buffer of memory for the DMQ in order to communicate with the node. If the size of the payload DMQ is chosen to be at least as large as the TCP window, the DMQ will automatically (due to it being a circular queue) contain the last window of payloads and so no additional memory is required. The bound on buffer requirements on the gateway is given by the combined bandwidth delay product of all connections through the gateway. For a maximum combined bandwidth of 1 Gbps, given that the average round trip time is likely to be less than 500 ms, there is a buffer requirement of 500 Mbits (i.e. 64 MB) for the entire cluster.8 As network speeds increase, this is likely to increase linearly with the

---

8This is the memory “in-use” (there may be more that is not used, as the DMQs may not be full) for a cluster in normal operation. Additional factors such as the number of connections and the TCP window size should also be considered for a full analysis.
4.3. Protocol Gateway and Retransmission

Normally (as shown in (a)) the stack copies data from the application before writing it over the DMQ (shown dashed). This copy is used for retransmissions. The proposed approach (shown in (b)) removes this copy by using the DMQ to retain the recently written data. The copy in the DMQ (at the gateway) is used for retransmissions.

Figure 4.2: Moving buffering for retransmission from the stack to the gateway
bandwidth as the average round trip time is likely to remain constant (or decrease). The only difficult part is reading and writing to this memory at the line speed of the network, but this can be achieved using current technology by having multiple memory banks in parallel and a wide bus to access them.

**Initiating a Retransmission**

There are two approaches to initiating a retransmission of a TCP segment:

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

The latter clearly simplifies the TCP on the server node and should result in faster retransmits. This is at the cost of complexity on the gateway as the gateway must identify when it needs to perform a retransmission. The former has the advantage that the gateway need keep no state for each connection other than the DMQs (which are needed anyway). For this reason it has been adopted for the CULT gateway. This partition of the protocol between host and network, with all the state being processed on the nodes, but the network performing buffering, is very promising. In particular, it does not suffer from the scalability problems that TOEs may do as they try to implement the state machines for all TCP connections at a single point in the network.

### 4.3.2 Implementation

The prototype gateway is, like the CULT stack, implemented at user level. There is no architectural reason for this other than ease of implementation and maintenance. Indeed, it could undoubtedly have achieved higher performance if purely kernel based as there is no “application” at user-level that must be involved, unlike the CULT stack on the nodes.
4.3. Protocol Gateway and Retransmission

The role of the gateway, from the perspective of TCP retransmission, is to take packets from the cluster, write them to the external network and retain a copy of the last TCP window of data from each stream.\(^9\)

To perform a retransmission the server node TCP sends a packet whose headers are identical to the (retransmitted) packet that would normally be sent at this point, with two exceptions:

- The payload (which the server node no longer has a copy of) is replaced with a TCP option field.

- The checksums (in the IP and TCP headers) are recomputed to ensure this is still a valid packet.

The gateway receives this in the normal way and upon noticing the TCP option field, it substitutes the correct payload and recomputes the checksums over the new packet before sending it out to the external network.

The TCP option is a total of eight bytes of payload, as shown in Figure 4.3. As with all other TCP options, the first byte identifies the type (a.k.a “kind”) of the option and the second the length. The following four bytes give the base address (relative to the DMQ) of the payload that needs to be substituted into the packet. The final two bytes give the length of payload that should be substituted.

![Figure 4.3: TCP retransmission option header](image)

---

\(^9\)Chapter 5 examines the other roles of the gateway: in particular on the incoming data path, with respect to demultiplexing and load balancing.
4.3. Protocol Gateway and Retransmission

Error Detection

As TCP retransmissions are performed by the gateway, this scheme brings an interesting variation to the problems of end-to-end reliability that were discussed in Section 2.4.2.

Although TCP checksums remain end-to-end for most packets, those that are retransmitted will have their checksum calculated by the gateway when it substitutes the payload. This will not, in most cases, cause problems, but for example, consider a packet that develops an error within the cluster, but this error goes undetected by the link-level CRC. The corrupted payload (assuming it is the payload and not the header that has the error) will be stored by the gateway. When the client receives the corrupt packet, an examination of the TCP checksum will (usually) reveal the fault, causing TCP to request a retransmission. The server will trigger the gateway to retransmit the packet, but the copy the gateway has is corrupt. In fact, because the buffering for retransmission by the gateway has released the server node from maintaining a copy of the data, there is no accessible good copy available. Worse still, because the gateway must recompute the TCP checksum on the retransmitted packet, it checksums the corrupt data. When this is received by the client the checksum will be valid, but the payload is still incorrect, causing it to erroneously accept corrupt data.

Although the above scenario is rare it still needs a solution. A number of possibilities present themselves:

- The approach taken by other schemes that suffer from a similar problem (e.g. TCP Offload Engines) is to choose a cluster network that is reliable (or at least has an acceptable level of reliability) such as Infiniband.\(^{10}\)

- A complete solution would be for the gateway to check every packet and acknowledge to the cluster node those that are correct. The cluster node would then have to buffer the packet for a cluster round trip time (which is likely to be much less than a full end-to-end round trip time) and retransmit those the gateway signals have become corrupted within the cluster. Although this would still be an improvement on having the end node retransmit all

\(^{10}\)Infiniband is discussed in more detail in Section 2.5.5.
corrupt packets, it would negate some of the benefits that the CULT architecture brings.

- An alternative is to have the gateway verify the checksums on all outgoing packets and if it finds that one has become corrupt it simply closes that connection. This guarantees that no incorrect data are transmitted.

Although the last option is crude, it is effective and in many cases acceptable. The rarity of this situation arising (it has never occurred in all the tests done during the course of this work, despite using an Ethernet based physical layer) coupled with the ability to remake the request if it should happen, results in it being a good way to recover. This technique is similar to that used by Infiniband to guarantee reliability, which also terminates a connection if an error is detected.

4.4 Summary

This chapter has highlighted how some changes to the way a TCP stack is implemented can have functional benefits such as the retransmission by the gateway. To judge whether these changes are justified, it is necessary to measure what effect they have on the performance of the stack. To do this, CULT has been implemented as a Linux shared library. This allows any standard application binary to make use of the CULT stack (in place of the normal operating system TCP stack) simply by setting an environment variable: \texttt{LD\_PRELOAD}. When this is done, the CULT library is loaded along with the application binary and it can then intercept the network access calls without disrupting the applications ability to access other types of socket. This allows a meaningful comparison (in the following section) as the applications used are entirely unmodified.

4.5 TCP/IP Performance

4.5.1 Test bed

To measure the performance of CULT and the effects of the changes presented in this chapter (and Chapter 3), a simple test bed consisting of three nodes is
used: the first as a cluster node, the second as an external node and the third which operates as the gateway. All three nodes are identical dual-processor Intel servers, with Xeon 2.4 GHz processors, Intel server (SE7501WV2) motherboards and 1 GB of RAM each. Each comes with two on-board Intel 82546EB Gigabit Ethernet Controllers on a dedicated PCI bus. In addition, each has a dual-port Level5 Networks gigabit NIC on one of the PCI-X slots. They are running RedHat 9, with Linux kernel version 2.4.20-21.9smp.

The servers are connected with an HP ProCurve 2708 switch. This is an un-managed eight port Gigabit Ethernet switch, with very low latency (< 2.5\,\mu s), rated throughput of 11.9 million pps and a switching capacity of 16 Gbps. It operates as a private network and so carries no other traffic other than that used for these experiments.

In the first instance, the cluster node was configured to use the CULT stack and gateway (as shown in Figure 4.4(a)). To allow comparison with existing technology, this was then compared to Gigabit Ethernet over the same links and network topology, but with a standard kernel stack running on all nodes and simple kernel-based IP forwarding used to bridge the two networks (as shown in Figure 4.4(b)). In both set-ups the external (i.e. non-cluster node) runs a standard kernel-based TCP/IP stack.

### 4.5.2 NetPIPE tests

To measure the performance, the popular and standard NetPIPE [100] benchmark was used. NetPIPE performs simple ping-pong transfers of varying size and so models to a reasonable approximation the sort of request-response traffic that many servers might have to cope with. It is widely used to measure the performance of computer networks.

The results of the NetPIPE benchmark, running over both the CULT network and the Gigabit Ethernet equivalent, are shown in Figure 4.5.

Figure 4.5 has a number of different lines for Gigabit Ethernet. This is due to the variability introduced by interrupt coalescence. The Intel drivers for the on-board Gigabit Ethernet NICs have a number of methods for reducing the load (and so increasing the performance) from interrupt handling. Section 2.4 discussed
4.5. TCP/IP Performance

(a) The CULT Test Network

(b) The Gigabit Ethernet Test Network

Figure 4.4: The test networks used to compare the CULT stack and gateway with a standard Gigabit Ethernet equivalent.
Figure 4.5: NetPIPE Performance of CULT and Gigabit Ethernet

Each line represents the average of ten runs of NetPIPE. The error bars show the maximum and minimum values recorded over those ten runs.
some of the problems with interrupt coalescence and this graph illustrates them. A significant part of this thesis has been how best to structure a stack at user-level; in CULT the use of interrupt moderation techniques is avoided and instead CULT focuses on the avoidance of the use of interrupts.

The main technique used by the Intel Ethernet drivers to moderate the number of interrupts is an Interrupt Throttle Rate (ITR). This restricts the NIC to not raising more than a certain number of interrupts per second and in normal operation the driver will attempt to set the threshold to an optimal value using built-in heuristics. The optimal value will strongly depend on the traffic pattern: for long streaming transfers, a low threshold will be sufficient, while for bursty or latency sensitive transfers a higher threshold will deliver much improved performance.

It was noticed that the default (dynamic) setting, where the driver should choose the optimal value, was delivering poor NetPIPE performance. This is shown by the “GigE (Dynamic ITR)” line in Figure 4.5. The Intel guide [46] to using the interrupt moderation facilities of the driver, recommends using the default dynamic setting, but suggests that “Linux-based operating systems appear to perform best with an interrupt rate between 1,000 and 8,000 interrupts per second”. These settings for the ITR were therefore used and plotted on Figure 4.5. However, despite in most cases representing a significant performance boost\(^{11}\) compared to the default dynamic setting, none approach the performance of the CULT network. In particular, for small transfers, the low latency characteristics of CULT results in it being able to deliver over double the performance of the best Gigabit Ethernet configuration.

Finally, it should be noted that despite NetPIPE averaging a large number of request-response transactions before delivering its results, there was significant variability in the reported Gigabit Ethernet performance. This is shown by the large error bars\(^{12}\) in Figure 4.5. CULT, on the other hand, delivered much more stable and predictable performance.

\(^{11}\)It should be noted that the default dynamic setting is likely to perform better when a different traffic pattern is presented and for this reason setting the ITR manually is not advisable in most situations.

\(^{12}\)These error bars represent the maximum and minimum samples measured, rather than standard deviation, as the data did not appear to be normally distributed.
4.5. TCP/IP Performance

### 4.5.3 Latency

To give an estimate of latency, NetPIPE measures the round-trip response time for each request it issues and then halves (to give a one-way latency) and averages this value. Table 4.1 shows the minimum, average and maximum of the ten runs of NetPIPE for both CULT and standard Gigabit Ethernet. These values correspond to a payload size of 12 bytes (the smallest used) for each packet.

As NetPIPE makes a large number of requests each second (sending the next one each time it receives a response) the latency values reported for Gigabit Ethernet will clearly be affected by the Interrupt Throttle Rate. The values shown in Table 4.1 are the ones recorded for the ITR set to 8000 (so representing the best available latency). This explains why the Gigabit Ethernet results are all just above 125 \( \mu s \).

To give an estimate of the latency without the interference of Interrupt Throttle Rate, the standard network tool “ping” has been used. This measures round-trip time in much the same way as NetPIPE (i.e. sends a request, which is echoed by the remote node and waits for the response) but as its request rate is much lower (one per second) it will not trigger the interrupt threshold. These values (based on 100 samples with 64 byte payloads and reporting latency not round trip time) are shown in Table 4.2.

<table>
<thead>
<tr>
<th>Network</th>
<th>Minimum (( \mu s ))</th>
<th>Average (( \mu s ))</th>
<th>Maximum (( \mu s ))</th>
<th>Mean Deviation (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>CULT</td>
<td>53</td>
<td>57</td>
<td>84</td>
<td>10</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>42</td>
<td>98</td>
<td>236</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4.2: Ping latencies

<table>
<thead>
<tr>
<th>Network</th>
<th>Minimum (( \mu s ))</th>
<th>Average (( \mu s ))</th>
<th>Maximum (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>CULT</td>
<td>42.7</td>
<td>44.9</td>
<td>45.6</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>133.0</td>
<td>142.0</td>
<td>150.9</td>
</tr>
</tbody>
</table>

Table 4.1: NetPIPE latencies
4.5. TCP/IP Performance

In both cases CULT is able to deliver lower latency than Gigabit Ethernet and again shows lower variation in its performance.

4.5.4 TCP Retransmission by the Gateway

A significant part of the restructuring of the TCP stack and the re-use of facilities provided by one component to make the role of another easier, is the use of the gateway to assist cluster nodes by buffering and retransmitting TCP segments.

To evaluate if this is a beneficial change, the overhead of the two approaches has been measured. To do this, the both (i) the cost to packets that are not retransmitted and (ii) the cost of retransmitting a packet is measured. To measure these accurately the same techniques that are discussed in Section 6.4 to profile the different timer routines are used. This involves using the rdtsc x86 instructions to read the processor cycle counter. To minimise the probe effect of taking measurements, these times are read into a pre-allocated circular buffer, which is then dumped to disk when the process quits, resulting in very fine-grained time measurements with only a few cycles overhead. These times can then be analysed off-line to produce frequency plots of the time distributions.

Non-retransmitted Packets

As most packets are not retransmitted, it is essential that they are not delayed as a result of these changes. In the case of packets which are not retransmitted, the comparative costs are (i) for the normal TCP/IP stack, the overhead of copying and freeing the payload of each packet on the server nodes and (ii) for the CULT stack, the cost of tracking the payloads and freeing the data from the DMQ on the gateway.

The cost of copying and freeing the payload (the traditional approach) is easy to measure: CULT is simply modified to do this at the appropriate points and profiling code inserted. The cost of the copy is likely to be proportional to the size of the payload. This is compared to the normal CULT code at the same points (which just allocates/frees the appropriate data structures, without performing a copy of the payload). The results of this are shown in Table 4.3.
4.5. TCP/IP Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Write Size (bytes)</th>
<th>Traditional time (ns)</th>
<th>CULT time (ns)</th>
<th>CULT Advantage (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc and copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>526</td>
<td>427</td>
<td>+99</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>807</td>
<td>572</td>
<td>+235</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1437</td>
<td>462</td>
<td>+975</td>
<td></td>
</tr>
<tr>
<td>1460</td>
<td>1357</td>
<td>635</td>
<td>+722</td>
<td></td>
</tr>
<tr>
<td>Dealloc and free</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>3498</td>
<td>2905</td>
<td>+593</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1298</td>
<td>1124</td>
<td>+174</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1083</td>
<td>1054</td>
<td>+29</td>
<td></td>
</tr>
<tr>
<td>1460</td>
<td>990</td>
<td>1184</td>
<td>-194</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Costs of copying payloads

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse Packet</td>
<td>66</td>
</tr>
<tr>
<td>Null</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 4.4: Cost to the gateway of examining each packet

The frequency plots from which these averages were obtained are shown in Figure 4.6.

For the CULT case, determining any additional cost of managing the DMQ is more complex. The vast majority of this overhead would be required even if the gateway were not being used to perform TCP retransmission (and indeed this is part of the attraction of this scheme), so differentiating the two parts is not simple. Despite the low-overhead measuring technique being used, the probe effect is still likely to swamp the simple pointer updates that are required for the DMQ. However, the parsing of each TCP packet to determine if a retransmission is required is a measurable cost that is wholly attributable to this method of performing retransmissions so this was measured and is shown in Table 4.4.

All the times shown are averages of 64576 samples. The traffic was generated using the “ttcp” application [65]. In Table 4.3, the traditional copy operations show considerably larger per-byte component than the CULT case, as expected. The negative per-byte trend for the “deallocate and free” operation is due to the amalgamation of writes into a single packet: for small writes, each “deallocate
4.5. TCP/IP Performance

Figure 4.6: Costs of copy and free operations
and free” will release many buffers (as they have been combined together into a single TCP segment). The smaller the write, the more they can be combined together and the more complex the combined free will be.

All but one of the results in Table 4.3 show CULT delivering an improvement,\textsuperscript{13} and once the costs of the two operations are combined, they all give a net gain to CULT. This is clearly dependent on the size of the write involved, but for these typical sizes (and on these servers) the gain is around 0.5-1.0 $\mu$s per write.

The additional cost at the gateway (shown in Table 4.4) is negligible in comparison. This table includes an entry for a “null” operation: essentially the time to do nothing and so a measure of the probe effect. Once this is taken into account, the corrected time to parse each packet is just 18 ns.\textsuperscript{14}

\textsuperscript{13}The reasons for this one rogue result that bucks the trend are not clear: it is most likely due to the libc \texttt{free()} routine being more efficient than my own memory pool for large chunks.

\textsuperscript{14}This correction is not made for the values in Table 4.3 as that case is mostly concerned with the differences between the two schemes, not the absolute values.
Load balancing is the art of dividing load among the servers in a cluster such that performance of the cluster is maximised. This chapter takes the stack and gateway from Chapters 3 and 4 and demonstrates how their structure and network interface provide a new technique for efficiently balancing the cluster.

5.1 Existing Approaches and Features

As discussed in Section 2.4.3 existing load balancing systems can be broadly grouped into two categories:

- **Centralised**: A single node in the system is responsible for dividing the work among the other nodes. This is typically combined with other functionality, such as a gateway or firewall, but represents a single point of failure, and may not scale well with the size of the cluster.

- **Distributed**: Work is initially assigned to each node either at random, or using a simple policy. The nodes are able to use some means of passing off work to other nodes when they are heavily loaded (or requesting work from
other nodes when they are lightly loaded). Although more complex, this approach has the potential to be more scalable and reliable.

There are some refinements to these two approaches, such as only sending connection requests through the front end (allowing all other traffic to go to and from the node that is doing the work directly) to reduce the load on it [42], or a combination of the two categories, where a front end will make the initial allocation, but nodes can still offload requests onto each other.

A survey of current commercially available load balancers reveals the following supported methods for balancing the cluster:

- **Round-Robin**: Each node is selected in turn, without any measurement taking place. This is cheap and simple, but sub-optimal and unresponsive.

- **Current Connections**: The node with the fewest current connections is used. It does not take into account the different workload that individual connections can represent.

- **Server Response Time**: The node with the lowest response time is used. Response time can either be based on past experience, or the request can be broadcast to all (or a subset of) available servers. The server which responds first is used, and the responses from the others are dropped (which clearly results in wasted effort).

- **Direct Measurement**: This involves measuring some resource on each node using an agent, and using the node which has the most of that resource. The added complexity of taking measurements on the server nodes is the main drawback.

- **Ratios**: Similar to Direct Measurement, but a ratio of resources is used, allowing dependency on a set of resources rather than a single one.

- **Content-Aware Persistence**: A repeated request is routed to the node that previously serviced it, resulting in improved locality. If a new (previously unseen) request is received some other technique must be used. A suddenly popular web page (for example) could result in the single node that serves that page becoming overloaded, while other servers stand idle.
5.1. Existing Approaches and Features

- **Client IP Address Hash**: The node is selected based on the client IP address. This results in the same server dealing with all requests from the same client. Failure of a single node will result in a subset of clients receiving no service, but the others will be unaffected.

Many of the drawbacks of these individual schemes can be mitigated by employing a combination of them.

### 5.1.1 Obtaining Load Measurements

If one of the measurement schemes is used, there are a number of metrics that can be used to determine the load of a system. CPU usage, number of outstanding requests, number of concurrent connections, and server response time are popular ones.

A distributor node acting as a centralised load balancer may be able to keep track of some of these (such as number of concurrent connections) by itself by maintaining state for each node in the network. However, a distributed system must explicitly gather the information from other nodes either by broadcast or by polling.

If broadcast is used to do this each node must periodically measure and broadcast its load to the others in the cluster. For this information to be useful the broadcast frequency must be high or there is a risk that the information may be stale and out of date. However, as the broadcast frequency is increased, the overhead of gathering the load information also increases. The broadcast approach must also avoid flocking behaviour, where all the nodes send their work to the least loaded server, immediately overloading it.

If instead polling is used, an overloaded server can request load information from the others when it needs it. The information should therefore always be relatively up-to-date. However, there is a delay (of the network round trip time) before the answer to the query is returned and it can be used to make a decision about where to send the work. It is also necessary to decide how many nodes to ask for load information [88], which clearly represents a trade-off between the accuracy of the decision and the cost of making it.
5.2. Proposed Alternative Approach

A distributed system also requires some method for the efficient hand-off of jobs from one node to the other such as Distributed Packet Rewriting [7] or Socket Cloning [90].

Locality-Aware [67, 6] load balancing is more sophisticated, and tries to take advantage of the reduced cost of serving many requests for the same data. By sending similar requests to the same node, the nodes should see improved locality and cache performance, and each naturally gravitates toward a dynamic specialisation (without having to be constantly re-configured to reflect changing patterns of requests).

5.2 Proposed Alternative Approach

The functionality of the CULT gateway has been extended to include the ability to balance load across the cluster. The primary context for this is load balancing web requests to a group of servers, but it could equally be applied to other applications.

The novel metric for load that has been chosen for evaluation is the available DMQ buffer space. In a conventional network this information (or its equivalent) would require an agent running on every node, distributing the current network buffer usage to all the others. In a DMQ network however the knowledge of the queue buffer usage is explicitly known by both communicating nodes as they are the flow control mechanism. The queue pointers (which together provide the buffer usage) are frequently updated with very little overhead. Also, as these pointers are required for the DMQ flow control, using them for load balancing as well imposes zero additional overhead on the server nodes.

An alternative to using the DMQ buffer usage would be to use information about the TCP window. The receiver advertises a window that limits how many segments can be transmitted before the sender must wait for further window updates. This, like the DMQ buffer usage, is information that is already passing through the gateway. However, there are an number of differences:

- In the DMQ case the buffer usage is described by the queue pointers, and because the two nodes are connected by a low latency network the updates can be considerably more frequent (every read and write).
5.2. Proposed Alternative Approach

- TCP relies on packets flowing in the reverse direction to carry the ACKs back to the sender.

- TCP window updates could also take longer to reach the gateway as in the absence of segments in the reverse direction acknowledgements can be delayed (for around 200 ms).

- TCP window information would have to be snooped from traffic passing through the gateway to discover the current window being advertised. However, this would require the gateway to parse the TCP packets, which would add significantly to its complexity.

- Protocols other than TCP would not be able to be balanced in this way, unless they also have a flow control window. The additional complexity of balancing on a protocol-by-protocol basis would be costly. The DMQ buffer usage applies to all protocols in the same way.

Although TCP window size bears many parallels to DMQ buffer usage, in practice it clearly would not make as good a metric for load balancing.

As well as being cheap to gather and up-to-date, a load indicator must be relevant. That is, it should provide an accurate indicator of the nodes ability to deal with a further request. This point is where existing load indicators often fail. For example, if the number of existing connections is used, no account is taken for how much traffic those connections are carrying. A single busy connection could provide much more load than many lightly used ones. Even if network buffer usage were measured in a conventional network, this would not give a reliable measure of how much progress the application is making. Because the network processing is carried out by the operating system in a different context (often at higher priority) than the application, network buffers can continue to be processed even if the application has stalled.

The CULT stack does not have this weakness. All the network processing is carried out in the same context as the application, and uses the application thread to do so. Therefore, if the application is busy, the received packets will not be processed as quickly, and the DMQ will start to fill up. It is also easy to
account network buffer usage to a particular application due to the connection-oriented nature of the DMQ. The tight coupling of the network processing to the application results in the network backlog being a much better indicator of load than would otherwise be the case.

This technique does not come without its weaknesses. It does not lend itself to the distributed approach to load balancing for example (as only the gateway has full knowledge of all the DMQs), and so suffers the issues of scalability and a single point of failure that are common to all centralised load balancers. This work does not include a means for overloaded nodes to hand-off work to other nodes; existing work does this well and it is hoped that it will be possible to avoid nodes becoming overloaded in the first place. It has also not attempted to provide any rigorous locality awareness, or ensure that subsequent requests are dealt with by the same server as this is orthogonal to the issue of the use of network buffer usage as a load metric.

5.3 Implementation

As the DMQs are already used for flow control and data transfer between the gateway and the server nodes, to implement this approach to load balancing requires little additional complexity.

In order to prevent the gateway having to iterate over every active connection (computing the DMQ buffer space and summing for each node) every time a new packet arrives this figure is incrementally maintained in response to changes in the queue pointers. While adding a small increase in complexity to a frequent data-path operation, this can be justified by the increase in scalability that it offers.

The load balancer can be configured to use either the number of bytes in the DMQ or the number of packets in the DMQ as the load indicator. Which of these is optimal is the subject of some debate; there is no hard-and-fast relation between the size of the payload of a request and the workload it will generate. However, the fact that the use of the number of bytes in the DMQ does remove “payload-less”

\footnote{All traffic relating to any one request, and any following ones on the same TCP connection will however always be routed to the same node by the gateway to ensure state for TCP connections does not have to be distributed throughout the cluster.}
packets (such as acknowledgements, which will generate very little additional load) from the equation means it does have some advantages.

5.3.1 Node Assignment

In the experiments that follow in Section 5.4 this new approach is compared to two more standard techniques: round-robin, and number of current connections. To compare them it is necessary to consider the complexity at the gateway of choosing which node to use next.

For round-robin, this is straightforward: simply pick the next node from the list of available nodes and use that one, looping when the end of the list is reached. This is $O(1)$ with respect to the number of nodes in the cluster.

If using the number of current connections or DMQ buffer usage, the choice of which node to use next is more complex. Each node has a “score” whose value describes its position according to the metric used. When a new connection arrives, the node with the lowest score must be used. There are two obvious approaches to determine which has the lowest score:

1. An ordered list of available nodes can be maintained such that the head of the list is always the least loaded machine. For example, whenever the buffer usage changes the relevant entry in the list would be moved to its new place in the list. The number of list operations would therefore be proportional to the number of changes in the score, but choosing the appropriate node when a new connection arrived would be $O(1)$ with respect to the number of nodes in the cluster.

2. Alternatively, rather than constantly maintaining this list, the set of nodes can be searched whenever a new connection request arrives. This is an $O(n)$ operation where $n$ is the number of nodes in the cluster.

Which of these is optimal will depend on the number of nodes in the cluster and the number of packets per connection (as in the DMQ buffer usage case this will be proportional to the number of times the score changes). For these reasons, both are studied in Section 5.4.2. It is likely that each connection will involve multiple
packets, and that the cluster will be of limited size (10s rather than 100s or 1000s of nodes).

5.3.2 Support for Standard Ethernet Networks

While the network used for this work is an excellent platform for research, it would be beneficial to demonstrate the effectiveness of this load balancing technique on a more traditional platform.\(^2\)

The DMQ load-balancing technique described above relies on the pointers in the DMQ to provide an indication of load. From this perspective a DMQ simply represents a connection where the buffer usage is known to both ends. As Ethernet is connectionless, implementing this on a standard Gigabit Ethernet network is not straightforward.

To overcome this, CULT can use a very simple connection-oriented Ethernet protocol to provide a channel for distributing the buffer usage information. This protocol is used on a cluster network for communication between the server nodes and the gateway. An additional header is placed after the standard Ethernet header. To avoid any non-CULT hosts on the Ethernet becoming confused should they receive one of these packets, a different Ethernet type is specified in the normal Ethernet header:

- ETH_P_CULT_OOB (0xaaaa): This is an out-of-band (OOB) message; i.e. a connection request, accept, or close.
- ETH_P_CULT_INFO (0xaaaab): This is a buffer usage information update only.
- ETH_P_CULT_DATA (0xaaaac): This is a buffer usage information update and an IP packet.

In the case of ETH_P_CULT_DATA the additional header is immediately followed by a normal IP packet. This prevents the information updates from requiring separate Ethernet frames, while the ETH_P_CULT_INFO type allows information updates in the absence of any IP data.

\(^2\)In addition, while waiting for supplies of the more capable hardware, it was necessary to create “proof of concept” implementations.
5.3. Implementation

<table>
<thead>
<tr>
<th>Ethernet Type</th>
<th>Operation</th>
<th>Dword 1</th>
<th>Dword 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH_P_CULT_OOB</td>
<td>Connect</td>
<td>Zero</td>
<td>Connection ID for reply</td>
</tr>
<tr>
<td></td>
<td>Accept</td>
<td>Connection ID from Connect</td>
<td>Connection ID for reply</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td>Connection ID</td>
<td>Zero</td>
</tr>
<tr>
<td>ETH_P_CULT_INFO</td>
<td>Information Update</td>
<td>Connection ID</td>
<td>Information Value</td>
</tr>
<tr>
<td>ETH_P_CULT_DATA</td>
<td>Information Update and IP Packet</td>
<td>Connection ID</td>
<td>Information Value</td>
</tr>
</tbody>
</table>

Table 5.1: Ethernet connection protocol header details

The additional header consists of two dwords. This is illustrated in Figure 5.1. These dwords are used as shown in Table 5.1.

To form a connection, one node sends a “Connect” ETH_P_CULT_OOB frame to the other and waits for an “Accept” in reply. The connection ID provided by the “Connect” is used in all subsequent frames to identify the connection. The zero connection ID is reserved to distinguish the different types of ETH_P_CULT_OOB frames.

Once a connection has been established, information updates and IP packets (in the form of ETH_P_CULT_INFO or ETH_P_CULT_DATA frames) can be exchanged until a “Close” is sent. A received “Close” should be acknowledged with a symmetric “Close” operation. No support for security or protection from spoofing (for example) has been provided; it is assumed that the cluster is a non-hostile environment. In addition, to be robust, the ETH_P_CULT_OOB messages need to
be reliable: i.e. they need to be retransmitted should they be dropped by a switch due to congestion on the Ethernet, or at least be given priority by switches to prevent them being dropped. This could be done using a similar technique to that used by the Address Resolution Protocol (ARP) [70] (i.e. simple retransmission), or by applying some of the existing work on QoS for Ethernet such as the IEEE 802.1p standard [43], or Ethernet Virtual Circuits.\(^3\)

The CULT stack can make use of either a full DMQ network, or this Ethernet based emulation, with only minor modifications to the lower layers. The physical network that has been used for these tests (from Level5 Networks) can be accessed using both a traditional packet-based interface and a memory aperture DMQ-style interface. This network has the advantage that even when being used as a standard Ethernet NIC it can be accessed from user-level, allowing us to use it directly from the CULT stack.

This protocol makes it possible to maintain the same buffer usage state at the gateway that is provided by the RDMA memory aperture network. It is still a low-overhead technique (although not as simple as the single dword write that the pointer update requires in the pure DMQ case), and is not dependent on a particular traffic pattern or higher layer protocol to be effective.

The performance measurements in the following section were made using the EtherFabric DMQ network, but it should be clear that this technique for load balancing can be easily applied to standard ethernet networks (among others).

5.4 Load Balancing Performance

5.4.1 Test bed

In order to gauge the performance impact of the changes in Chapter 5 (the use of network buffer usage as a load balancing metric) a more complex test bed has been developed.

This consists of a cluster of servers, intended to be a good representation of a typical set that might be deployed commercially. The cluster is made up of six

\(^3\)Ethernet Virtual Circuits was a project at AT&T Laboratories Cambridge Ltd, which unfortunately was not published before the closure of the lab in 2001.
5.4. Load Balancing Performance

dual-processor Intel servers, identical to those used in Section 4.5. The servers are connected, as before, with an HP ProCurve 2708 switch.

One of the six servers is configured to operate as the gateway, as illustrated in Figure 5.2. This bridges to a second gigabit ethernet network, which is interconnected with a second HP ProCurve switch. This connects to a number of client PCs to provide load for the servers. The clients are Dell PCs, with Intel desktop motherboards, single Pentium 4 3.2 GHz processors, and a single on-board Intel 82540EM Gigabit Ethernet Controller. They are running Suse 9.0, with Linux kernel version 2.4.21-226-default. As in the previous experiments, the networks used do not carry any other traffic.

Figure 5.2: The test bed network

5.4.2 Node Assignment

The first aspect considered is the complexity at the gateway of choosing which node to use next. As described in Section 5.3.1 the new buffer usage technique
5.4. Load Balancing Performance

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-Robin</td>
<td>290</td>
<td>297</td>
<td>306</td>
<td>351</td>
</tr>
<tr>
<td>Connections</td>
<td>835</td>
<td>959</td>
<td>2998</td>
<td>36978</td>
</tr>
<tr>
<td>Connections</td>
<td>294</td>
<td>302</td>
<td>296</td>
<td>353</td>
</tr>
<tr>
<td>Buffer Usage</td>
<td>995</td>
<td>1116</td>
<td>2954</td>
<td>36210</td>
</tr>
<tr>
<td>Buffer Usage</td>
<td>298</td>
<td>292</td>
<td>294</td>
<td>209</td>
</tr>
</tbody>
</table>

Table 5.2: Mean time in ns to determine which node to use

is compared to two standard approaches: round-robin and number of current connections.

The profiling code based on the rdtsc x86 instructions is again used to measure the time taken to do these operations. The code profiled was the function invoked when a new connection arrives to find the best (according to each algorithm) node to use, and assign the new connection to that node.

The tests were performed using a simulation of a cluster to allow much larger cluster sizes than were physically available to be measured. Each test primed the simulation with 1000 connections, and then proceeded to pick one at random, close it, and open another. The time to assign the new connection was measured and this was repeated $2^{15}$ times. Cluster sizes of 1, 10, 100 and 1000 nodes were simulated.

For the “number of current connections” and “buffer usage” metrics two different approaches are profiled (as discussed in Section 5.3.1): constantly maintaining a sorted list such that the head of the list is the best node, and searching the list for the best node whenever a new connection needs to be assigned.

The results of these experiments are shown in Table 5.2.

It is clear from this that as expected the round-robin algorithm is $O(1)$ with respect to number of nodes in the cluster. The slight increase for the cluster size of 1000 could possibly be attributed to a cache effect as the data set grows in size.

The two search-based variants (of the current-connections and buffer-usage schemes) show a strong dependence on the cluster size. Perhaps more surprisingly, the sort-based variants (again of the current-connections and buffer-usage schemes) also appear to be roughly $O(1)$ with respect to number of nodes in the
cluster. This can be explained as follows:

- In the case of the current-connections metric, the list of nodes consists of a set of very similar, if not identical, values (the load balancer acts to ensure this). For example, with 999 connections distributed over 10 nodes, it might be expected to be [99, 100, 100, 100, 100, 100, 100, 100, 100, 100]. When the 1000th connection is added, the node with a load of 99 will be chosen, but it will not need to move position in the list as it is still correctly ordered for its new load (100). Hence the sort is a null operation, and does not depend on the length of the list. However, when a connection is removed, and a node moves from a load of 100 to 99, it will need to move position (to the head of the list), and this may depend on the list length. The cost of sorting on connection removal is shown in Table 5.3.

- In the case of the buffer-usage metric, addition and removal of load both require a sort operation as the load has more variation than the current-connections metric so it does not achieve the “perfectly balanced” situation described above. In addition, as the loads do not change when a connection is added or removed (assuming the buffer usage of both new and completed connections is zero) the list does not require sorting at this time. However, for the sorted buffer-usage algorithm, the change in load (and so the list sorting) happens when a packet within the cluster is read or written. It is therefore necessary to include this cost: it is shown in Table 5.4.

These results enable the trade-off between maintaining a sorted list (and its associated per-packet cost) and searching the list to be evaluated. For the buffer-usage metric, in a cluster of size 100, the sorted list variant will outperform the unsorted list variant unless the number of updates (and so sorts) exceeds 55 per connection.\(^4\)

\(^4\)There are always 999 connections at this point, as one of the thousand is removed by the simulation every time one is added.

\(^5\)The exact relationship will depend on the sorting technique used: for these experiments a bubble-sort approach is used as this performs well on a nearly sorted list: these lists are nearly sorted as only a single element (the one that has just had its load changed) is ever in the wrong place.

\(^6\)Using the values from the “Cluster Size = 100” columns in Table 5.2 and Table 5.4: 
\[
\frac{(\text{Difference in assignment})}{(\text{Time to update})} = \frac{(2954 - 294)}{48} = 55.4
\]
5.4. Load Balancing Performance

<table>
<thead>
<tr>
<th>Remove connection sort cost</th>
<th>Cluster Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>178</td>
<td>230</td>
</tr>
</tbody>
</table>

Table 5.3: Mean time in ns to sort list after connection removal

<table>
<thead>
<tr>
<th>Update only</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update and sort</td>
<td>69</td>
<td>66</td>
<td>66</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>87</td>
<td>114</td>
<td>1283</td>
</tr>
<tr>
<td>Difference</td>
<td>14</td>
<td>21</td>
<td>48</td>
<td>1205</td>
</tr>
</tbody>
</table>

Table 5.4: Mean time in ns to update load indicator

Discussion

These micro-measurements show that the buffer-usage metric (although introducing some additional complexity when compared to round-robin) represents a similar amount of overhead at the gateway as the system based on number of current connections. All of the algorithms tested are sensitive to the traffic pattern used. To gauge what, if any, difference they make to the overall cluster performance a macro-measurement of the cluster as a whole (rather than the gateway in isolation) is required.

5.4.3 Httperf

To measure the overall performance of the cluster and the effect load balancing techniques have the Httperf [63] benchmark is used. This benchmark generates load at a particular rate, and measures how many of the requests can be satisfied by the server. The workload (i.e. the pages served) is taken from the SPECweb99 [26] benchmark suite: this is designed to be representative of real web traffic, including pages of a variety of sizes as well as requests for both dynamic and static content.

The server used is lighttpd (version 1.3.6) [51], chosen largely because it is a
simple, efficient and easy to use web server. Lighttpd is run on the (up to) five servers in the cluster, with the sixth configured as the gateway as illustrated in Figure 5.2. To serve dynamic web pages lighttpd is configured to use a FastCGI extension. This means it does not have to fork a new process for each dynamic request, and instead the dynamic requests are all served by a single process in an infinite loop. Communication between the web server and the FastCGI process is performed using a Unix domain socket.

### Uniform Clusters

The cluster was first configured so that all five of the servers were identical. The gateway was switched between the different modes of load-balancing operation and tests were done with each.

To measure the efficiency of the load balancer in this scenario, the response times both for TCP connections and HTTP requests are measured using httperf. The workload chosen issues five HTTP requests on each TCP connection and tries to achieve a TCP connection rate of 1000 connections per second. The TCP connection lifetime measurements in Table 5.5 provide statistics about how long it took to complete each set of requests. These are shown graphically in Figure 5.3. Table 5.6 provides TCP connect time measurements (the time for a TCP connect call to complete) and HTTP request time measurements (the time between sending the first byte of a single HTTP request and receiving the first byte of the HTTP reply).

The uniform configuration of the cluster should favour round-robin over the buffer-usage metric. The uniform nature of the connections should favour the

<table>
<thead>
<tr>
<th>Load Balancer</th>
<th>Max Concurrent Connections</th>
<th>Min (ms)</th>
<th>Avg (ms)</th>
<th>Max (ms)</th>
<th>Median (ms)</th>
<th>Std Dev (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-Robin Connections</td>
<td>37</td>
<td>4.0</td>
<td>10.4</td>
<td>98.1</td>
<td>9.5</td>
<td>6.1</td>
</tr>
<tr>
<td>Buffer Usage</td>
<td>20</td>
<td>3.8</td>
<td>9.2</td>
<td>20.1</td>
<td>8.5</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>4.2</td>
<td>9.0</td>
<td>26.8</td>
<td>8.5</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 5.5: TCP connection lifetimes for different load-balancing metrics
Figure 5.3: Distribution of TCP lifetimes for the different load-balancing metrics
5.4. Load Balancing Performance

<table>
<thead>
<tr>
<th>Load Balancer</th>
<th>TCP connect (ms)</th>
<th>HTTP request (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-Robin</td>
<td>1.2</td>
<td>1.7</td>
</tr>
<tr>
<td>Connections</td>
<td>1.1</td>
<td>1.5</td>
</tr>
<tr>
<td>Buffer Usage</td>
<td>1.1</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 5.6: Response times for different load-balancing metrics

The current-connections metric over the buffer-usage one. Perhaps unsurprisingly, the current-connections metric does seem to minimise the TCP connection lifetimes. However the results do not bear out the round-robin prediction: the buffer-usage metric delivers better results.

This can be attributed to the way the buffer-usage algorithm for choosing the “best” server works: it will always choose the first server unless that server has higher buffer usage than the others. If it does, the second server will be chosen (likewise, unless it has higher buffer usage than the rest) and so on. This means that while the first server is able to keep up and process the requests as fast as they come in (and so keep its buffer usage low) it will often be chosen by the load balancer, while other servers will receive few or no requests. Only when the first server reaches saturation will the load balancer start to divert requests to the others. As a result, the first server is kept “hot” (it will likely have the right files and code in its caches) and can process each request faster than a server that has a delay between each one.

Non-Uniform Clusters

The round-robin metric makes an implicit assumption that each server in the cluster is equivalent and each is able to handle the same number of requests. As the buffer-usage metric measures the load on each server, it does not make this assumption. To demonstrate what happens when this assumption is invalid, a cluster was configured with two classes of servers. The first class consists of servers left as normal, while the second class have both SMP and hyperthreading disabled. This restricts the second class to a single CPU, where the first has 4 virtual CPUs (2 physical CPUs). Httperf was then used to generate a load that a set of the nor-

\[\text{Weighted} \] versions of this metric to account for differences in servers are possible.
5.4. Load Balancing Performance

<table>
<thead>
<tr>
<th>Load balancer</th>
<th>Servers</th>
<th>HTTP 200 (%)</th>
<th>HTTP 503 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-Robin</td>
<td>Both normal</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>One normal, one disabled</td>
<td>47.9</td>
<td>52.1</td>
</tr>
<tr>
<td>Buffer Usage</td>
<td>Both normal</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>One normal, one disabled</td>
<td>72.7</td>
<td>27.3</td>
</tr>
</tbody>
</table>

Table 5.7: Percentage of requests satisfied for uniform and non-uniform clusters using different load balancing metrics

normal servers could serve without problems, but a set of the disabled servers would struggle to deliver. The load consisted of requests for dynamic web pages (i.e. involved running a script).

This load was then presented to two cluster configurations: (i) a pair of normal servers, and (ii) one normal server and one disabled server. Httperf returns (among other things) a count of the HTTP return codes. If the server is able to serve the page it returns 200 (“OK”) along with the requested page but if it is too busy it returns 503 (“Service unavailable”). The relative percentages of these gives a good indication of the performance of the server, which in this case is directly attributable to the accuracy of the load balancer. These results are shown in Table 5.7 and graphically in Figure 5.4.

Non-Uniform Workloads

From the previous tests in Section 5.4.3 it is clear that the buffer-usage metric delivers similar (i.e. to within experimental error) performance to the connection-based metric. While it is good that in those cases the proposed use of buffer usage is able to deliver as good performance as the most popular current technique, to justify the additional complexity it is necessary to consider a case where it will outperform connection-based load balancing.

To do this requires a workload that breaks the assumption that more connections equates to more load. It is difficult to imagine connections that do not add to the load of a server in any way and so load will always be monotonically increasing with number of connections. However, not all connections are equal. Some connections will, in a realistic cluster, add considerably more load than others.
5.4. Load Balancing Performance

Figure 5.4: Comparison of round-robin and buffer-usage metrics

Each metric was tested on two clusters: one uniform (all servers identical) the other non-uniform (half the servers had SMP and hyperthreading disabled).
5.4. Load Balancing Performance

For example, a server may be providing both static and dynamic content. A request for static content can be satisfied much more easily than a complex script or database request, which may entail considerable processing. However, as these complex requests will take longer to serve (almost by definition) the connections they are tied to will last longer, which will in turn result in the number of connections at that server node increasing. This will force the load balancer to correct it. Therefore, statistically (over the long term) these more complex requests will still be distributed throughout the cluster, but if they are few in number (and particularly over short time scales) they could still become concentrated on a single node.

Idle connections present other problems [23] for connection-based load balancing. A workload that has significant numbers of persistent HTTP connections [60] could demonstrate this. These idle connections will not reduce a server node’s ability to take another connection, but will add considerably to the load balancer’s opinion of how loaded a node is. A node that has a disproportionately high number of idle connections will therefore be ignored, at the expense of the others. Again, over the long term, the connection-based approach will even this out, but over short time scales there could be significant differences. The buffer-usage metric would not count an idle connection (assuming all the traffic has been processed) when computing a node’s ability to handle a new request.

In summary, generating realistic workloads [58, 37] that are scientifically meaningful and also illustrate the problems with connection-based workloads is difficult. These workloads are likely to be transient, but it is how a load-balancer responds to such short-term spikes that is its most important feature. In the situations described, a direct measure (such as buffer usage) from the server will prove more reliable than inference at the load balancer. In particular, as the buffer-usage metric is fine grained and low latency, it should be good at responding to such situations.

Others have tried to address the problems of connection-based load balancing by distributing load at the granularity of HTTP requests, rather than TCP connections [90]. This is in some senses orthogonal to the buffer-usage proposition, as buffer usage could also be used at that layer.
5.4.4 Bandwidth Overhead

To allow the gateway to use the buffer usage of each node as a load balancing metric, the nodes must update the gateway with their buffer usage. As this is required for the flow control of the distributed message queue, using this information does not use any additional bandwidth. However, to avoid the need for the gateway to sum the pointers in all the message queues for all connections on each node, a running total is maintained. This could be done by the gateway by watching for changes in the queue pointers, calculating the difference with the previous value, and adjusting the running total for that node appropriately. However, it is arguably simpler for the node to keep its own running total and transfer it via the network to the gateway whenever it changes. Assuming it changes on every packet read, and that full MTU (1500 bytes) packets are being used, this single dword write is a 0.27% overhead. If smaller packets are used, the percentage overhead would increase, but the nodes could also employ a “hold-off” technique such that the buffer usage information is only propagated when it has changed significantly, rather than on every packet read.

5.5 Summary

The use of DMQ buffer usage as a metric for load balancing has been shown to give performance at least as good as current popular techniques and in many cases better. Buffer usage, when coupled with a stack at user level using the same thread of control as the application, provides a real indicator of how busy the application is. It combines the benefits of direct measurement (relevant and accurate information) with inference techniques (low overhead and up-to-date), but does not require additional complexity beyond that needed for data transfer via a DMQ.

As a result it provides accurate balancing of a cluster that scales well to high workloads, automatically deals with non-uniform clusters and non-uniform workloads and should cope well with spikes in load.

The same approach has been shown to be possible even in the absence of support for DMQs.
Chapter 4 examined how to structure a stack for efficient implementation at user level. This chapter extends that by considering how to implement TCP timers within that structure.

6.1 Timer Implementation Problems

TCP uses timers to cope with inactivity. Whenever an action or response is expected from a remote node, the local TCP will set a timeout to recover in case that action or response is not received. For example, when a segment is transmitted, TCP sets a retransmission timer which will resend the segment if it is not acknowledged.

In total there are seven different timers:

**Connection-establishment timer:** set when a SYN is transmitted, and aborts the connection if no response is received within 75 seconds.

**Retransmission timer:** set when data are transmitted and retransmits the data if no acknowledgement is received.
6.1. Timer Implementation Problems

**Delayed-acknowledgement timer:** set when data are received that do not need to be acknowledged immediately. If, after 200 ms, the acknowledgement is still pending (i.e. it has been unable to piggy-back on an outgoing data segment) it is then sent.

**Persist timer:** set when the remote node advertises a window size of zero, thus preventing any data being sent. If the window has not been opened when the timer expires, 1 byte is sent in case the window update was lost. (Window updates, like acknowledgements, are not sent reliably).

**Keep-alive timer:** expires after 2 hours of inactivity (if requested by the application) and sends a special segment to keep the connection open.

**FIN WAIT 2 timer:** set if the connection is in the FIN WAIT 2 state, and cannot receive any more data. If after 11 minutes 15 seconds a FIN has not been received the connection is dropped.

**TIME WAIT (or 2MSL) timer:** set when a connection enters the TIME WAIT state and expires after twice the Maximum Segment Lifetime (MSL). The state for the connection is deleted, allowing the socket to be reused.

The duration of the delayed-acknowledgement and persist timers depends on the measured round trip time of the connection. The reasoning behind all these timers is discussed in detail by Stevens in TCP/IP Illustrated. [109, Chapter 25]

### 6.1.1 Historical Constraints

When TCP was specified in RFC 793 [71] in 1981 the support for time measurement in operating systems, when compared to today, was poor. TCP was also designed to be easily portable to a large number of operating systems and so made very few assumptions and demands about what was available, even if some systems could offer much more. In addition, TCP did not need as accurate time measurements as it does now because the time intervals being measured were larger: slower networks resulted in lower bandwidths and longer round trip times.\(^1\)

\[^1\]For example, the original TCP specification bounds a measured network RTT to be greater than 1 second. As this is used to time retransmission, it can lead to unnecessary delay in modern
As a result, to implement all the timers TCP only requires that two functions are called periodically: (i) the \textit{fast timer} function is called every 200 ms and (ii) the \textit{slow timer} function every 500 ms. TCP uses these two periodic “ticks” to schedule and check all the timers described above, as well as measuring round trip times.

As network speeds have increased, this relatively coarse-grained ability to measure time and schedule actions has become increasingly noticeable and although operating systems can now measure time very accurately with very little overhead (particularly in the kernel), TCP has not adapted to make full use of this facility (although some modifications such as TCP Vegas \cite{13} do go some way to improving matters). The problems that this leads to are widely known and there have been many proposed solutions. These were discussed fully in Section 2.7.

\subsection{Timer Implementations}

These historical constraints have to some extent shaped the way that most TCPs implement timer support.

\subsubsection{List Searching}

Setting and clearing timers is a common operation (each sent and received segment will involve at least one or more timer operations) and so this must be cheap: usually just involving setting a flag in the protocol control block. However, this means that on each of the fast and slow clock ticks the list of connections must be searched to discover which require attention. This has time complexity of $O(n)$ where $n$ is the number of connections. This section uses the delayed-acknowledgement timer to illustrate the drawbacks of this approach.

As well as the scalability issue, searching the list of connections each clock tick has a second detrimental side effect. Timers are, in a user-level context, most easily implemented as a separate thread. Great effort has been expended to avoid the need for multiple threads per-connection in the CULT stack. If a separate thread were used for timers, many of the benefits gained from single threading networks, which will commonly have RTTs of less than 100 ms. This is one of the problems addressed by TCP Vegas \cite{13}.\)

\end{document}
6.1. Timer Implementation Problems

would immediately be lost: locking would be required for access to the stack to ensure that the state for each connection is only used by a single thread at a time.

**Inaccurate Delay**

When the list of connections is searched for those that have delayed acknowledgements it is not known when, other than after the last fast-timer tick, this acknowledgement was delayed; i.e. it could have been waiting for anything between 0 ms and 200 ms. This has two side-effects: (i) an acknowledgement could be sent unnecessarily if it had only been waiting for 0 ms; and (ii) an acknowledgement could be delayed for much greater than the RTT of the network if it had been waiting for 200 ms.

The TCP specification suggests that at least every other acknowledgement should be sent; i.e. if there is a request to delay an acknowledgement and one is already pending on that connection, an acknowledgement should be sent straight away.\(^2\) This means that on a reasonably busy connection (even if it is half duplex and so acknowledgements will never be able to piggy-back on data) it should never be necessary to send a delayed acknowledgement on a timer tick. However, in practice, on average every other fast timer tick will result in one being sent.\(^3\)

6.1.3 Time at User Level

Most TCPs are kernel based. As CULT is a user-level stack, it has a different set of problems when trying to efficiently implement timers. Accurate time measurement is a common operation within the kernel and so is well supported: on Linux, for example, the “jiffies” variable is incremented whenever the timer interrupt occurs (typically 100 times a second). The kernel also provides “task-queues” or “tasklets” and “kernel timers” that can be used to schedule execution of code at a later time - ideal for implementing TCP’s timers.

As with many operating system resources, access to such features from user level requires a system call (e.g. `gettimeofday()`) which is relatively ex-
pensive. Solutions to this such as Soft Timers [3] are able to reduce this cost by executing the event handler at certain (common) entry points into the kernel, when the overhead is already being taken. There is also the possibility of using a real-time scheduler to ensure that a process is executed at a specific time.

The approach taken by CULT is to use the free running processor cycle counter which is accessible on x86 systems using the “rdtsc” assembler instructions. This has very low overhead and provides high-resolution (of the order of nanoseconds) time information. It does require calibration however, to gain an accurate measure of time in more standard units (seconds rather than processor cycles).

### 6.2 Cross-Thread Scheduling

An interesting way to view this problem of efficiently implementing the fast timer is to consider it as a scheduling problem. Threads (or processes) can be in many scheduling states: (i) *Runnable* when it is only prevented from running because the processor is busy with something else; (ii) *Running* when it is actually executing on the processor; and (iii) *Waiting* when it is waiting for something (some I/O to complete, or a lock to be released for example) and so cannot currently continue.

A typical state diagram for the life cycle of a process is shown in Figure 6.1.

![Figure 6.1: Scheduling States of a Process](image)

Processes enter the **Waiting** state when they need some other process or task
to complete an action before they can continue. There is generally no mechanism for the event that the processes is waiting to complete to be changed once it is in the Waiting state, or for a process to move from Runnable to Waiting without going through Running. This is because, in general, the process itself must decide when to enter the Waiting state and the only way for it to change its mind about that is for it to be executed: i.e. it must move into the Running state to determine that it needs to be Waiting.

In the case of user-level TCP timers, a timer thread would usually be in the Waiting state, waiting for 200 ms to pass. At the end of this time it will become Runnable and eventually Running when it reaches the head of the Runnable queue. However, because there it usually no work for it to do, it will quickly return to the Waiting state. In other words, it is polling.

It is possible to determine in advance if the timer thread has any work to do: the timer thread must be executed if there are any acknowledgements that have been delayed for more than 200 ms (or whatever threshold time is set). In addition, this calculation can be performed by any other TCP thread, thus preventing the need for the timer thread to be scheduled only to discover it should go back to Waiting. This could be implemented as follows:

Each TCP connection has either a time at which it would like the fast timer to be run (200 ms since it delayed an acknowledgement) or no requirement for the fast timer (if it does not currently have any acknowledgements pending). If any of the times for the connections that have delayed acknowledgements are less than the current time, then the timer thread needs to be run. However, these times are constantly shifting as acknowledgements are sent and other acknowledgements are delayed. As a result, the time at which the timer thread should next become Runnable is constantly updating. However, as mentioned above, there is currently no mechanism whereby the condition that a process is waiting on can be updated. If support for this were available, it would mean one group of threads or processes (the data-sending threads) preventing another (the timer thread) from becoming Runnable, which would be a new scheduling concept.

Processes do, of course, already influence when others can be Runnable through, for example, condition variables (and it may be possible to produce the desired effect through the use of these), but there is currently no way to explic-
6.3. CULT Timer Architecture

Itally alter the condition that another process is waiting on; only whether or not that condition is true. Condition variables would also require one of the other threads to be active at the right time in order to signal the condition variable as met, whereas the suggestion above would still operate correctly if none of the data threads are currently running. Scheduler Activations [1] (where the kernel schedules processors to a group of threads, which then use a user level scheduler to divide the processor among them, so allowing for an efficient N:M threading model) are more promising and could be extended to perform this role.

To support this would require significant changes to either the Posix thread library or the kernel (depending on which scheduler is being used) to provide the necessary API. However, due to the complexity of this and the fact that for CULT it is hoped that the timer thread can be avoided completely for active connections (as discussed below) this has not been done. It would however constitute a very interesting area of further work and as scheduling is so key to many aspects of operating systems, it could have many other applications.

6.3 CULT Timer Architecture

This dissertation presents a different approach to TCP timers, developed to counter the problems described. Namely: (i) the inaccurate delay of acknowledgements (Section 6.1.2); and (ii) the scalability and thread-consequences of searching lists of connections by timers (Section 6.1.2). CULT uses a system of timer buckets to provide both upper and lower bounds on the delay of acknowledgements and a partitioning of connections to avoid the problems presented by the necessary timer thread. These are described in the rest of this section.

6.3.1 Fast-Timer Buckets

Timer buckets are a means of providing the upper and lower time limits on a delay. CULT uses them for the delayed acknowledgement timer. To do this, there are two buckets. Events that require a delay are marshalled into one of them. The target bucket (i.e. the one that new events are placed in) is swapped periodically. Each bucket is emptied with the same periodicity as the switch of the target bucket, but
out of phase. This is illustrated by Figure 6.2.

![Figure 6.2: Timer Buckets](image)

The difference in phase between switching the target bucket and emptying that bucket is responsible for ensuring a non-zero lower limit; i.e. there is a delay between a bucket no longer being the target, and that bucket being emptied. The upper limit is, obviously, enforced by the fact that the buckets are regularly emptied.

In the case of delayed acknowledgements, it is convenient to set the lower limit at 50 ms and the upper limit at 150 ms. This maintains the average delay from the standard 0-200 ms range, but both reduces the maximum delay and introduces a guaranteed minimum delay to ensure that piggy-backing of acknowledgements is encouraged. To achieve these limits, the period of the bucket switching and emptying is 200 ms, with the emptying 90° (i.e. 50 ms) out of phase, as shown in Figure 6.2.

### 6.3.2 Lazy Implementation

This dissertation focuses (in part) on how to efficiently implement protocols. This aspect of TCP needs a way to regularly switch between the two buckets and empty them appropriately, ideally without requiring a separate thread to control it.

This has been achieved using a lazy approach: whenever a data thread performs an timer operation, or an operation that is blocked, it determines if a timer bucket action is required (i.e. switch or empty bucket).
Each bucket is represented using a simple counter. It records the number of delayed acknowledgements that are currently pending. When an acknowledgement is delayed, the current bucket’s counter is increased. When an acknowledgement is sent, the counter is set to zero.

Four variables are used to control the buckets: (i) the current target bucket to use if delaying an acknowledgement; (ii) the time at which the next switch of the target bucket should take place; (iii) the bucket to examine when next “emptying” a bucket; and (iv) the time at which the next bucket emptying should take place. They are updated as follows:

**Current Target Bucket:** Whenever an acknowledgement is delayed (just before the timer bucket counter is increased) the time at which to make the next switch is compared to the current time and if it has passed the buckets are switched. This has the side effect that, because it is done lazily, the buckets do not switch exactly every 100 ms and in the absence of any acknowledgements being sent a bucket could span considerably more than that. This is, however, safe in that it only happens to an empty (inactive) bucket: as soon as something tries to add to a bucket, the switch occurs and the “next switch time” is updated accordingly.

**Bucket to Empty:** Whenever a bucket is emptied, a “struct timeval” timeout structure is set to 100 ms from the current time. This structure is passed to any blocking operation involving that connection and when one times out (once 100 ms has passed) the timer handler is executed without requiring a separate thread. The timer handler examines the current bucket, sends any acknowledgements that have been delayed and updates the timeout structure.

This approach separates the timing of the bucket switch and the bucket emptying and so the two may drift, resulting in one not falling exactly half-way between the other. This effectively alters the phase difference between them and is not ideal: it will alter the limits on the time for which acknowledgements are delayed.

---

4 For performance, the processor cycle counter (rather than a system call) is used to obtain a measurement of the current time.
6.3. CULT Timer Architecture

However, the fact that they are not in phase is enough to ensure that there is usually sufficient delay to provide a non-zero lower limit on the timers, and so enable piggy-backing of ACKs to take place more often.

6.3.3 Implications

Although at first glance the benefit of not sending a few unnecessary acknowledgements is small, this has wider implications. It has been suggested that TCPs’ timers’ frequencies should be increased to allow for more accurate time measurement and lower delays in responding to inactivity on modern networks. As the period of the timers decreases, the proportion of acknowledgements that would occur immediately before a clock tick would increase and so the number of unnecessary acknowledgements would also rise. By using this bucket approach a gap will always be maintained between a delayed acknowledgement being requested and the timer that sends it if nothing else has.

By maintaining a bucket for each connection, the scheme also removes the need to search all connections on each timer tick: each connection is now responsible for administering its own delayed acknowledgement timers.

This comes at the cost of increasing the complexity of requesting and cancelling a delayed acknowledgement, mostly as a result of the need to compare times to determine when to switch buckets. An implementation in C of the new algorithm on the Intel x86 architecture (making use of the “Read Time Stamp Counter” processor cycle counter for measuring time) requires an additional 28 assembler instructions (assembled without optimisations on a dual Pentium 4 Xeon architecture). Whether this trade-off is beneficial will depend on the frequency of the fast timer, the number of active connections and the traffic pattern experienced; the performance of the different timer approaches is discussed in Section 6.4.

While these timer bucket changes can be implemented on their own, additional benefits can be realised by extending the concept of making each data thread responsible for the timers of that connection to avoid the possibility of the data thread coming into conflict with the timer thread. The following section demonstrates how a partition of the connections can reduce the need for locking (that the presence of the timer thread would normally mandate) on the data path.
6.3.4 Timer Threads

The need for a separate timer thread arises from the requirement that some timers occur during periods when the application will not perform any operations on that connection.\footnote{For example, the \texttt{TIME WAIT} timer is used to clean up connections after they have been closed.} As a result, the data thread will not be executing any protocol code and so it can not be used to perform timer operations. This in turn leads to locks being required for both the timer thread and the application thread to ensure exclusive access to a connection.

As the timer thread is required for some cases where the data thread is not active, to avoid locking it is necessary to ensure that only one of the threads will attempt to use a connection at any one time. To do this CULT ensures that those connections which are active (so likely to receive attention by a data thread) are not touched by the timer thread. The timer thread only looks after the timers of those connections which are inactive.

The split of connections is managed by maintaining lists of those in each group. Timers on connections in the \texttt{CLOSED}, \texttt{LISTEN}, or \texttt{TIME WAIT} states are managed by the timer thread. The rest are managed by the application data threads.

The application data thread manages timers as described in Section 6.3.2: it compares the current time against the next required timer whenever it enters a blocking operation. If 100 ms have passed since the last “tick”, the fast timer (and if 500 ms have passed the slow timer) routine is called, but it only operates on that connection.

The timer thread operates much as it would in a normal environment, waking every 100 ms, looping over the list of connections it is responsible for and calling the fast and slow timer functions for each one.

When the connection lists are manipulated they must be locked to ensure consistency, but this only happens at connection setup and tear-down, and so is not on the data path.
6.3.5 CULT Timers Summary

The changes presented here have the following properties:

- No locks are required on the data path as only a single thread can operate on each connection at any one time. This represents a significant efficiency saving.

- Clock ticks on active connections do not require iteration over all connections to determine which need attention.

- Timers on active connections occur with lower priority than data operations. This is because timers occur when an operation would block, i.e. when a data operation cannot immediately complete. This is justified by the use of timers to react to inactivity: if there is data activity, then the timers are not required. This property is of particular interest to a loaded server where checking timers (but not doing anything useful as nothing is required) could take a significant portion of CPU time.

- The accuracy of timers on active connections is reduced and we cannot guarantee they will be called in a timely manner, although the probability of them being unduly delayed is low.

The impact of these changes on the performance of the stack is explored in the following section.

6.4 Measurements and Results

Section 6.3.1 proposed a scheme that would reduce the number of delayed acknowledgements sent (by encouraging piggy-backing) with the side effect of increasing the cost of queueing an acknowledgement.

The exact trade-off between increased complexity of queueing an acknowledgement versus decreased likelihood of sending an unnecessary delayed acknowledgement is complex. It is highly dependent on the traffic pattern, number of (and any interaction between) connections, the application that is using the stack, the
cost of obtaining a mutex and many other possible variables. The suggested modifications will clearly not improve matters in all cases. This section attempts to quantify these trade-offs.

### 6.4.1 Cost of Queueing an Acknowledgement

The proposed bucket scheme for the delayed acknowledgement timer increases the complexity of queueing an acknowledgement. This is a common operation (almost every received packet will result in this operation being called), so even a small increase in complexity could be detrimental.

To measure this effect, profiling code was inserted into the function that queues and dispatches acknowledgements. This profiling measured and recorded the number of processor cycles that were spent in the added section of code and the number of cycles spent in the old (non-bucket) equivalent. The latter of these is just a single bitwise OR operation to set a flag. The results of this are shown in Figure 6.3.

![Figure 6.3: Time Taken to Queue a Delayed ACK](image)
As these are very small sections of code there is a significant “probe-effect”: the timings include the time spent measuring the time. However, this is constant for both versions, so a comparison is valid. The times (in seconds) are calculated offline from the processor cycle count using the bogomips measure. The test used to obtain these results was a simple “ttcp” benchmark [65], running on a Dual Intel P4 Xeon 2.4 GHz server.

From Figure 6.3 the added complexity of dealing with buckets adds 34 ns to each acknowledgement request. This overhead will be taken each time the quantity of unacknowledged data exceeds twice the TCP Maximum Segment Size (MSS) and the frequency of this will clearly depend on the traffic pattern. Consider a single connection using the full theoretical 1 Gbps of bandwidth and a MSS of 1460 bytes, it will happen every 11.7 us. The additional overhead will therefore amount to 0.29% of a single CPU. In practice the overhead is considerably less than this as the theoretical maximum bandwidth is rarely achieved and so the acknowledgements are requested less frequently.

6.4.2 Cost of Fast Timer Routine

The benefit of the bucket scheme is avoiding the need to call the fast timer routine. To measure the saving this represents a small piece of test code was created. This test wakes up every 200 ms and iterates over a list of “connections”. Each connection is locked with a mutex and a flag is checked, in the same way that would happen in the fast timer function. In this way the test simulates the work needed to check a set of connections for a delayed acknowledgement, but not the work involved in dequeueing and sending any delayed acknowledgements found. As a result, the test measures just the work that has been avoided by the bucket scheme. It is also possible to easily vary the number of connections in a way that would be difficult to do on a “live” TCP stack.

The average time per iteration of the connection list and the CPU usage this represents is plotted in Figure 6.4 for different list sizes. The times plotted do not include the overhead of sleeping and waking this process every time the list

---

6 This analysis has ignored the bandwidth taken up by the network protocol headers.

7 These “connections” are just data structures; there is no data transfer involved.
needs to be checked. The times were calculated using both in-line profiling and the UNIX “time” utility (taking the ratio of user CPU time to elapsed real time to calculate the CPU percentage).

Figure 6.4(b) shows that the point at which the list iteration becomes more expensive than the bucket scheme (assuming the cost of 0.29% of a CPU as suggested in Section 6.4.1) is approximately 4000 connections, depending on which technique is used to measure the time taken. This may seem high, but it should be noted that this is the worst case for the bucket scheme and an ideal case for the connection list iteration scheme. Other factors which would increase the cost of the list iteration (such as the thread switch overhead) were not taken into account. Also, if the period of the fast timer was reduced in the future (or this approach used for something other than TCP where the frequency is higher), this would result in the bucket scheme becoming more favourable. The bucket scheme does not have a direct dependence on the number of connections (it is capped by the number of acknowledgements sent, which is itself a function of the number of packets received) and so represents a more scalable solution.

6.4.3 Reduction in Data Path Locks

The major benefit of this work is the removal of the need for many of the locks on the data path. A conservative estimate of the performance impact of this is as follows: a stack operating at full 1 Gbps network bandwidth, transferring data to or from the application 1 KB at a time, will involve 122070 read or write operations each second. If the bucket scheme has avoided a single lock/unlock pair of operations on each of these calls (measured above to take approximately 150 ns) the total saving will amount to 1.8% of a CPU, ignoring other factors. This dwarfs the overhead (0.29%) of the bucket scheme.

6.4.4 Discussion and Comparison

There are currently no widely used user-level TCP stacks to compare to the approach suggested here, therefore (although it has been noted that timers at user-level are a different implementation problem to those in the kernel) this section
6.4. Measurements and Results

Figure 6.4: Overhead of iterating over a list of connections

6.4(a) shows the cost in units of time. 6.4(b) translates this into fractions of a CPU. Note the log-log scales.
examines two of the most common (open-source) kernel TCP stacks: Linux and FreeBSD.

During the period of time that this work has been carried out, the Linux kernel TCP has improved markedly. At the time of writing, the latest Linux kernel (2.6.5) uses the generic (i.e. not specific to TCP) kernel timer support. This is based on a hierarchical timing wheel [102] and an evaluation (of an earlier, but largely similar, version: Linux 2.5) showed it performed and scaled well [68]. Timing wheels (in their simplest form) represent an ordered circular list of events, with each element in the list containing the events that should occur at a particular time. On each tick of the clock, the next element in the list is examined and any events present are processed. This scheme is $O(1)$ (with respect to the number of timers) for setting timers, clearing timers and performing each clock tick, but does require that all timers are set for periods less than a fixed upper bound (to prevent wrapping around the circular list). The timing wheel scheme has been extended [15, 29] to remove the need for the fixed upper bound and dynamically size the “wheel”.

This approach has similarities with the bucket scheme presented here: it replaces the need to search lists of connections on each timer tick with an individual, per-connection timer. This avoids the need to bucket delayed acknowledgements, as each connection is dealt with individually, but is only possible because of the ease (and low cost) with which kernel timers can be modified from within the kernel. This would be hard to replicate at user-level without the ability to change the scheduling condition that a process is waiting on, as outlined in Section 6.2. It also, in the forms currently used, requires a regular clock tick to service the wheel. At user level this lends itself to a separate thread, which is something CULT needs to avoid. It may be possible to adapt it to use a lazy evaluation and so allow it to be used in-line by the data threads as proposed in Section 6.3.2

Timing wheels are also used by the FreeBSD, implementation and an implementation for the BSD UNIX version of TCP has demonstrated their scalability [27]. In FreeBSD each connection has a separate delayed acknowledgement timer and timers are managed centrally using a timing wheel. The timer is reset to 100 ms (rather than 200 ms) each time an acknowledgement is requested as a result of received segments, if there is no delayed acknowledgement timer in
progress. This condition means that delayed acknowledgements are sent for every other received segment, rather than waiting for the outstanding unacknowledged data to exceed twice the maximum segment size. The timer is cancelled (if in progress) when a segment is sent.

As with Linux, this is made possible by the ability to manipulate timers with low overhead in the kernel. However, the fact that both Linux and FreeBSD have opted for an increased complexity timer set/clear operation and individual timers for each connection in order to achieve greater scalability to some extent justifies the goals and approach taken in this paper.

6.5 Summary

In summary, this approach to timers represents a mechanism for periodically calling events without requiring a separate thread of control. This allows it to be more efficient as there is no additional thread switching and reduced locking overhead. It does however require wrappers for all blocking operations and does not provide absolute guarantees about the timing of events.

This approach has been used to support the delayed-acknowledgement timer in CULT, with the added benefit that a finite lower bound on the timer has been provided, ensuring piggy-backing of acknowledgements on data packets is encouraged.
CHAPTER 7

Conclusions

This dissertation set out to demonstrate how server clusters can benefit from a user-level TCP stack and what roles in protocol processing the network components can efficiently perform. This chapter summarizes the results and contributions that have been made toward this goal and outlines some interesting areas for further research.

7.1 Contributions

Chapters 1 and 2 provide an introduction and background survey of related work. Chapters 3 presented a new architecture for a user-level network stack. It outlined how protocol processing could be bypassed, allowing the application to interface more closely with the network layer. This enabled connections at multiple levels to have a closer association and so simplified the demultiplex operation. The CULT structure dispensed with dedicated threads for network protocol processing and instead used the application thread. This removed the overhead of context or thread switching for network access and gave greater accountability of resource usage to the application for network operations.

Chapter 4 demonstrated how this architecture could provide data transmis-
sion and delivery operations to the application. It developed a novel scheme for buffering payloads in case of retransmission that did not require a copy. This was achieved instead by reusing functionality of the network: the Distributed Message Queue and the gateway. The gateway also performs the role of demultiplexing incoming packets. Both of these substantially reduce the overhead of network access at the server without an equivalent increase in overhead at the gateway.

The implementation of a network stack resulting from Chapters 3 and 4 (CULT) was shown to deliver approximately double the bandwidth achievable using traditional network stack architectures and hardware. Latency was also improved: CULT was measured using NetPIPE to have a two-hop (through the gateway) average latency of 44.9 $\mu$s vs 142.0 $\mu$s for normal Gigabit Ethernet. Using Ping, CULT measured 57 $\mu$s vs 98 $\mu$s for Gigabit Ethernet. The bandwidth and latencies also show substantially reduced variation under CULT.

Chapter 5 extended the role of the gateway from demultiplexing and retransmission to load balancing. The use of network buffer usage as a metric for load balancing was presented and evaluated. It was shown to outperform a round-robin approach and match the performance of a connection-based metric, with connection lifetimes reduced by 15% and 2% respectively for uniform workloads and cluster configurations. Quantitative experiment and qualitative discussion outlined how it would further outperform traditional schemes when more realistic workloads and non-uniform clusters were considered.

Chapter 6 integrated TCP timer support into the CULT architecture. A dedicated thread was required to provide timing guarantees to inactive connections, but data path locks were avoided by a lazy in-line approach for active connections. This led to slightly increased cost for setting and clearing timers (an additional 34 ns per operation). The avoidance of list searching through the use of timer buckets resulted in increased scalability in terms of number of concurrent connections. Finally, the avoidance of data path locks was shown to result in even greater savings.

7.2 Further Work

Work using CLAN and EtherFabric continues at the Laboratory for Communication Engineering. Although the lab does not have the resources to develop the
Further Work

there are a number of questions raised by this work that could provide further interesting research.

Cross-Thread Scheduling The concept of cross-thread scheduling, where one thread or process can directly control another thread or process’s scheduling state by updating the condition on which it is waiting, was introduced in Section 6.2. This, as a novel scheduling concept, could have wide ranging implications and applications. It would benefit from a more in-depth analysis and implementation.

Timers The approach to timers in Chapter 6 could have applications in other areas. It represents a scheme for periodically calling events without requiring a separate thread of control where both upper and lower bounds on timing are required, but the events can tolerate some inaccuracy or jitter. This could, for example, be of particular interest to software for embedded systems that do not have an operating system.

Application Programming Interface Section 3.3 described how some aspects of the sockets API and OS functionality, such as exec() have not been fully implemented for CULT. In most cases this is more an implementation challenge than academic research.

Producer-Follower Event Distribution Section 3.3.4 introduced the producer-follower technique for dispatching events to a group of threads. This warrants further analysis to demonstrate its scalability and relevance to other areas that require this functionality.

Signalling and Out-Of-Band Messaging A current weakness of the CULT support for ordinary networks (such as Gigabit Ethernet) is one of connection setup and out of band messaging (outlined in Section 5.3.2). There is considerable room for improvement here, providing reliable and timely delivery of out of band packets to allow creation of the network-level connections. Ideally this would be done (as with CLAN) with assistance from the hardware, but even there connection setup can be costly. One suggestion would be to use a single long-lived TCP connection to transfer the network-level
connection requests from gateway to node (and vice-versa). This is similar to the approach taken by some routers, which use a long-lived TCP connection to share route information via BGP. The gateway and nodes could also maintain pools of unused network-layer connections and recycle them for new requests. Alternatively, preallocating the distributed shared memory apertures, then binding them at a late stage to a new connection could reduce the latency of connection setup between gateway and server.

**Load Balancing** Finally, the use of network buffer usage as a load-balancing metric is worthy of further study. It would be interesting to see how this metric performs in a live cluster with real (and therefore dynamic and unpredictable) workloads.

It would also be beneficial to demonstrate its utility over other RDMA networks and how it can be integrated with new standards such as DDP [8].

### 7.3 Outlook

As network speeds are currently increasing faster than processor speeds, the overhead of accessing the network using traditional techniques will further dominate CPU usage in coming years.

This dissertation has presented a number of techniques to reduce this overhead. By reusing functionality that is already present in the system, rather than simply moving the overhead elsewhere (as is the case with alternatives such as TCP Offload Engines), real benefits can be delivered to applications.
Bibliography


[31] Frederick Douglis. Transparent process migration in the sprite operating system. Technical Report CSD-90-598, University of California, Berkeley, 1990. [Referenced on page(s) 72]


[34] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proceedings of Hot Interconnects V*, August 1997. [Referenced on page(s) 18, 29, 34]


[41] Phil Howard. VRB - Virtual Ring Buffer. [http://phil.ipal.org/freeware/vrb/](http://phil.ipal.org/freeware/vrb/). [Referenced on page(s) 80]


[44] IEEE. Information Technology – Portable Operating System Interface (POSIX) – Part xx: Protocol Independent Interfaces (PII), P1003.1g. Institute of Electrical and Electronics Engineers, March 2000. [Referenced on page(s) 18]


[56] Level 5 Networks Ltd. http://www.level5networks.com/. [Referenced on page(s) 29]


[61] Jeffrey C. Mogul. TCP Offload is a dumb idea whose time has come. In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003. [Referenced on page(s) 46, 87]


[65] Mike Muuss and Terry Slattery. tcp. [Referenced on page(s) 100, 137]


[71] Jon Postel. Transmission control protocol. RFC 791, 1981. [Referenced on page(s) 125]


[83] Cesar A. F. De Rose, Reynaldo Novaes, Tiago Ferreto, Fabio A. D. de Oliveira, Marcos E. Barreto, Rafael B. Avila, Philippe O. A. Navaux, and Hans-Ulrich Heiss. The Scalable Coherent Interface (SCI) as an Alternative for Cluster Interconnection. [Referenced on page(s) 33, 36]

[84] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Pstorage over ip: When does hardware support help? [Referenced on page(s) 88]


Pattern Languages of Programs Conference, 2000. [Referenced on page(s) 73]


