High-Throughput Local Area Network Access for INMOS Transputers

R.M.A. Peel

A thesis submitted to the University of Surrey for consideration for the degree of Doctor of Philosophy

Department of Electronic and Electrical Engineering
University of Surrey
Guildford, Surrey
United Kingdom

14th August, 1995
Abstract

This thesis presents the design of an Ethernet local-area network interface for embedded transputer systems. It is based upon parallel software which manages the TCP/IP family of protocols, passing packets between a single transputer, which connects to the network, and application processes which run on an arbitrary number of other transputers. The different layers of the protocol processing - Ethernet control, IP and TCP are all performed in separate parallel processes. Extra routing processors, arranged in a tree configuration, provide access to the lower IP and Ethernet layers from as many TCP and application processes as desired.

Investigation of the processor utilisation and channel throughput of each of the parallel processes has led to the rejection of hardware-assistance in the form of a complex shared-memory, multi-processor architecture. Instead, a double pipeline of processes, running on a small pipeline of transputers, communicate exclusively using the transputers' serial links. This scheme is shown to provide good load balancing and to be a cost-effective way of exchanging traffic between a transputer application and a user process running on a high-performance workstation at data rates of over 950 kbytes/second - almost the entire available bandwidth of a 10 Mbit/sec Ethernet. All software is written in the occam programming language.

As well as presenting the design of the protocol software, the thesis includes performance measurements and reports on two applications which were built upon the initial work. These are a networked implementation of the INMOS Iserver, which allows access to transputers from anywhere on the network, and an embedded instrumentation system which pre-processes data from an ion microbeam and passes part-analysed results to a conventional workstation for display, archiving and user control of the experiment.
Acknowledgements

The work reported in this thesis has been supported in many ways by very many people. In particular, I would like to thank:

- Dr. B.M. Cook for his motivation, good advice and onerous proof-reading duties over many years;
- Drs. R.G. Forbes and A.V. Shafarenko for their guidance as my supervisors;
- Project students Piers Shallow, Kjetil Hamre and David Millen for their contributions to various items of hardware and software;
- INMOS Limited and Sundance Multiprocessor Technology Limited for timely donations of hardware and software which made my task far less painful;
- The developers of the freely-available Linux operating system and GNU text processing tools, and Christoph Niemann for his B004 device drivers, which have all performed perfectly when most needed during the latter stages of the project;
- and finally, my parents, sister, colleagues and students for tolerating long periods of intermittent communication while parts of these technical developments and this thesis came together.
Table of Contents

Abstract ................................................................................................................................. 1
Acknowledgements .................................................................................................................. 2

1 Introduction and Motivation ............................................................................................. 9
  1.1 Local Area Network Access for Transputers ............................................................... 9
  1.2 Networked Compiler Environments ............................................................................ 10
  1.3 Choice of Programming Language ............................................................................. 11
  1.4 Ethernet and the TCP/IP Networking Protocols .......................................................... 11
  1.5 Structure of this Thesis .............................................................................................. 12

2 Background and Literature Survey .................................................................................. 13
  2.1 The INMOS Transputer ............................................................................................ 13
  2.2 The Occam Programming Language .......................................................................... 15
  2.3 Occam Idioms and Methods ....................................................................................... 18
    2.3.1 Classes of parallelism - Geometric, Algorithmic and Process Farming ................ 18
    2.3.2 Deadlock Freedom ............................................................................................... 21
    2.3.3 Multiple Buffering ............................................................................................... 24
    2.3.4 Fair Selection ....................................................................................................... 25
  2.4 The TCP/IP Family of Protocols ................................................................................ 26
    2.4.1 The OSI Model ................................................................................................... 27
    2.4.2 The TCP/IP Protocols ....................................................................................... 29
  2.5 Transport Protocol Implementations on Sequential Machines .................................... 33
  2.6 Parallel Implementation of the TCP/IP Protocols ....................................................... 37
  2.7 Summary of the Literature Survey ............................................................................. 46

3 Research and Experimental Objectives ........................................................................... 48

4 A Parallel TCP/IP Software Architecture for Transputers ........................................... 50
  4.1 Detailed Code Structure ............................................................................................. 54
    4.1.1 Low-Level Ethernet Control Software .................................................................. 54
      4.1.1.1 AMD LANCE Control Software ...................................................................... 56
      4.1.1.2 The Effect of Direct Memory Access on Link Throughput .............................. 59
      4.1.1.3 Fujitsu NICE Control Software ...................................................................... 62
    4.1.2 Multiple Buffering Strategy .................................................................................. 64
    4.1.3 The IP Process ..................................................................................................... 66
    4.1.4 The TCP Router Processes ................................................................................... 68
      4.1.4.1 A Multiply-buffered Multiplexer ..................................................................... 72
      4.1.4.2 A Multiply-buffered Demultiplexer .................................................................. 73
      4.1.4.3 Routing Decisions in the Root and Tree Routers ............................................. 74
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The basic protocol-processing pipeline</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Block diagram of a single INMOS transputer</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Geometric Parallelism</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Algorithmic Parallelism</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Processor Farm Configurations</td>
<td>21</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The Client-Server process model</td>
<td>22</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Overwriting Buffer and Prompter processes</td>
<td>23</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Occam code for two multiple buffer implementations</td>
<td>25</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Occam code for two fair alternation implementations</td>
<td>26</td>
</tr>
<tr>
<td>Figure 10</td>
<td>The OSI Model Protocol Structure</td>
<td>27</td>
</tr>
<tr>
<td>Figure 11</td>
<td>The TCP/IP Protocol Structure</td>
<td>29</td>
</tr>
<tr>
<td>Figure 12</td>
<td>The INMOS B300 - hardware architecture</td>
<td>38</td>
</tr>
<tr>
<td>Figure 13</td>
<td>The INMOS B300 - Software architecture</td>
<td>39</td>
</tr>
<tr>
<td>Figure 14</td>
<td>The Karlsruhe Transputer-based Protocol-Processing Architecture</td>
<td>40</td>
</tr>
<tr>
<td>Figure 15</td>
<td>The IBM Parallel Protocol Engine</td>
<td>42</td>
</tr>
<tr>
<td>Figure 16</td>
<td>The full TCP process architecture</td>
<td>52</td>
</tr>
<tr>
<td>Figure 17</td>
<td>A Minimal set of TCP/IP processes</td>
<td>54</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Communications during the boot process</td>
<td>56</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Low-level Occam interface to the LANCE controller</td>
<td>57</td>
</tr>
<tr>
<td>Figure 20</td>
<td>The NICE controller process</td>
<td>62</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Occam code transfers transmit packets into the NICE hardware buffer</td>
<td>63</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Occam code transfers received packets from the NICE hardware buffer</td>
<td>64</td>
</tr>
<tr>
<td>Figure 23</td>
<td>The Prompter Mechanism</td>
<td>65</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Occam code for two prompter implementations</td>
<td>66</td>
</tr>
<tr>
<td>Figure 25</td>
<td>The IP Process Buffering</td>
<td>67</td>
</tr>
<tr>
<td>Figure 26</td>
<td>TCP Packet Distribution using a Ring Structure</td>
<td>69</td>
</tr>
<tr>
<td>Figure 27</td>
<td>TCP Packet Distribution using a Mesh Structure</td>
<td>69</td>
</tr>
<tr>
<td>Figure 28</td>
<td>TCP Packet Distribution using a Tree Structure</td>
<td>70</td>
</tr>
<tr>
<td>Figure 29</td>
<td>TCP Packet Distribution using a Crossbar Switch</td>
<td>71</td>
</tr>
<tr>
<td>Figure 30</td>
<td>A Tree-Structured TCP Routing Scheme</td>
<td>72</td>
</tr>
<tr>
<td>Figure 31</td>
<td>A Multiply-buffered multiplexer</td>
<td>73</td>
</tr>
<tr>
<td>Figure 32</td>
<td>A Multiply-buffered demultiplexer</td>
<td>74</td>
</tr>
<tr>
<td>Figure 33</td>
<td>The distributed TCP routing mechanism</td>
<td>74</td>
</tr>
<tr>
<td>Figure 34</td>
<td>The occam code structure of the main loop in the TCP process</td>
<td>77</td>
</tr>
<tr>
<td>Figure 35</td>
<td>The TCP Process, its channel interfaces and shared buffers</td>
<td>78</td>
</tr>
<tr>
<td>Figure 36</td>
<td>The UDP Process</td>
<td>82</td>
</tr>
<tr>
<td>Figure 37</td>
<td>Processes and channels referred to in the Deadlock-freedom Discussion</td>
<td>84</td>
</tr>
</tbody>
</table>
Figure 38 Checksum Algorithm - throughput for all Ethernet packet sizes
Figure 39 Checksum Algorithm - throughput for small packets
Figure 40 Testing two TCP processes
Figure 41 Testing the Link Performance
Figure 42 Concatenating two row communications into a single buffer
Figure 43 Link Performance Measurements
Figure 44 Throughput of the IP-to-IP and TCP-to-TCP filter processes
Figure 45 Measuring the TCP Process Performance
Figure 46 TCP Throughput as a Function of Receive Buffer Size
Figure 47 TCP Throughput as a Function of Transmit Buffer Size
Figure 48 The effect of various window buffer sizes on the TCP throughput
Figure 49 Variation in TCP Throughput against transmitted Segment Size
Figure 50 Variation in TCP Packet Rate against transmitted Segment Size
Figure 51 The Effect of Transmission Delays on the TCP Process
Figure 52 Testing the TCP Root Router Process Performance
Figure 53 Throughputs of the Root Router and the Tree Router Processes
Figure 54 Exercising the TCP routing processes using multiple streams
Figure 55 Tree Router Performance - TCP towards IP
Figure 56 Tree Router Performance - IP towards TCP
Figure 57 Root Router Performance - TCP towards IP
Figure 58 Root Router Performance - IP towards TCP
Figure 59 Harness for the IP Performance Measurements
Figure 60 Throughput of the IP Process
Figure 61 Configurations for Multiprocessor Performance Measurements
Figure 62 Unidirectional traffic across stack 3, showing checksum locations
Figure 63 Unidirectional traffic across stack 5, showing checksum locations
Figure 64 Testing the Transmit Performance of the Ethernet Module
Figure 65 Transmission rates on Ethernet for differing packet sizes
Figure 66 Process Configurations for the Ethernet Measurements
Figure 67 Four transputers yield the highest Ethernet throughputs
Figure 68 A Transputer hosted in a Personal Computer
Figure 69 A Networked Transputer Facility
Figure 70 A larger Networked Iserver configuration
Figure 71 The networked Iserver process structure - whilst servicing requests
Figure 72 A Networked Transputer Facility (full process layout)
Figure 73 The Telnet Status Scheme
Figure 74 Networked Iserver - the effect of increased SP packet sizes
Figure 75 DTR1 Block diagram
Figure 76 DTR2 Block diagram
Table of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>The Impact of DMA on a T222 Transputer's Link Throughput</td>
<td>60</td>
</tr>
<tr>
<td>Table 2</td>
<td>The Impact of DMA on a T225 Transputer's Link Throughput</td>
<td>61</td>
</tr>
<tr>
<td>Table 3</td>
<td>Application-to-Application throughput using Multiple Transputers</td>
<td>115</td>
</tr>
<tr>
<td>Table 4</td>
<td>TCP/IP Performance across Ethernet to a SparcStation20</td>
<td>124</td>
</tr>
<tr>
<td>Table 5</td>
<td>TCP/IP Performance across Ethernet using faster transputers</td>
<td>127</td>
</tr>
<tr>
<td>Table 6</td>
<td>Iserv server baseline hardware performance</td>
<td>139</td>
</tr>
<tr>
<td>Table 7</td>
<td>Iserv server performance - loading a large program</td>
<td>139</td>
</tr>
<tr>
<td>Table 8</td>
<td>Iserv server performance - latency of empty packet transfers</td>
<td>140</td>
</tr>
<tr>
<td>Table 9</td>
<td>Iserv server performance - latency of 512-byte packet transfers</td>
<td>141</td>
</tr>
<tr>
<td>Table 10</td>
<td>Ion Microbeam - Costs of collecting maps and spectra</td>
<td>161</td>
</tr>
</tbody>
</table>
CHAPTER 1

1. Introduction and Motivation

The INMOS transputer is a microprocessor which incorporates parallelism and inter-process communication deep inside its architecture and instruction set. The transputer architecture allows programs to be written as sets of separate parallel processes which communicate in order to share information. These processes may then all be run on a single transputer, under the control of a hardware process scheduler, or they may be distributed across several processors and communicate using high-speed serial links.

Transputers are therefore particularly suitable for use in embedded control and instrumentation systems, where the sub-division of the overall task into small parallel processes provides a natural programming model. In addition, these parallel processes may be run on several processors, if necessary, to obtain the required performance and real-time responses.

It became clear that attaching embedded transputer systems to local area networks would provide a means of mass data transfer and remote control previously unavailable, yet conceptually very similar to native inter-transputer communication.

1.1. Local Area Network Access for Transputers

The simplest method of attaching transputers to a local area network would be to attach them to a networked host processor, and to add extra software to the host to relay messages back and forth between the network and the transputers. One of the disadvantages with such a scheme is its high latency - the delays associated with each stage of moving and processing the messages accumulate unacceptably. Flow control typically requires polling from the transputer to the host, increasing traffic at that interface, unless a complex interrupt-driven direct memory access scheme is implemented between the processors. Another disadvantage is the single-tasking nature of the most common host software architecture (the DOS or Windows operating environment running on the IBM-compatible personal computer), which considerably hinders the servicing of both network and transputer inputs. Furthermore, unless the message relaying software is incorporated within the operating system kernel, the overheads of continually re-scheduling user-level code will be excessive. These issues were explored in early work by myself [Peel88] and subsequently by Luigi Rizzo [Rizzo93].

Many of the deficiencies of providing local area network access for transputers could be avoided by running the protocol processing code on the target transputers themselves, and providing a
hardware network interface to these processors. This has been the main goal of the research presented in this thesis. Rather than just porting conventional sequential protocol processing code to a single transputer, considerable effort has been expended in parallelising the task. A double pipeline of processes is used at the lower layers of the protocol-processing hierarchy; separate processes are used for the transport-layer processing of each stream of network traffic. Refer to figure 1 for an illustration of a simple pipeline.

![Protocol Processing Pipeline](image)

*Figure 1 - the basic protocol-processing pipeline*

Enhancements to this basic scheme are required to support network access for a number of application processes, rather than just for one, and for different protocols - in particular UDP as well as TCP. In addition, it will usually be necessary to spread the load of the application and protocol processing amongst a number of transputers, thereby increasing throughput further. This processor parallelism introduces the need to route partially-processed network packets between transputers, and to overcome limitations imposed by the restricted number of communication links on each transputer.

Chapter 4 explains the design decisions which have been made to accommodate these requirements. The work is unique and provides exceptional performance and connectivity for clusters of embedded transputers.

### 1.2. Networked Compiler Environments

For historical reasons, transputer software development platforms have usually run on a transputer which is plugged into a host processor - commonly a PC. Application software is edited within the host environment, and the source code is then piped to the target transputer hardware where it is compiled, with all the compiler's filestore requirements being met by the host. The INMOS Transputer Development System (TDS) [INMOS90] carries this concept even further - the editing environment is closely integrated with the compiler and both run on the transputer. Compiled transputer programs fall into two categories - those which run alone and those which require access to filestore and display facilities. The latter must be run on clusters of transputers which are serviced by a host system, just as the compiler was above.
There are considerable attractions to providing networked transputers, capable of running both compilers and the final application programs, within a central resource. The transputers become accessible to all processors on the local area network, rather than just the one in which they are installed. In addition, users do not have the burden of locating a computer containing suitable transputer boards, or of finding that all such host facilities are occupied by users who are not actually utilising the transputer component.

A networked transputer system, which supports an arbitrary number of target transputers and provides them with access to host file services, is described in chapter 5.

1.3. Choice of Programming Language

The programming language which best supports communicating processes on the transputer is occam, devised by INMOS at the same time that the transputer architecture itself was defined. Occam incorporates most of the features of conventional (sequential) programming languages, and adds exciting new ones such as process parallelism, inter-process communication, built-in timing facilities and greater security through type checking of values being communicated and usage checking of variables being shared between parallel processes.

Not surprisingly, occam is therefore used for all of the transputer code which has been written to support my research. Some features of occam, such as its requirement for static memory allocation, have had an impact on the code design, and these are discussed.

1.4. Ethernet and the TCP/IP Networking Protocols

The underlying communication structure used in this research is Ethernet [Digital80]. Ethernet is a common-bus style of local area network which runs at a basic speed of 10 Mbits/sec. It thus provides performance of a similar order to the 10 Mbits/sec and 20 Mbits/sec links provided by transputers - a little too fast for the former and comfortably accommodated by the latter. The underlying hardware allows many users to broadcast messages to any of their neighbours. Low-level algorithms resolve the contention which occurs when several stations attempt to transmit at the same time. Individual Ethernet segments may be connected together by repeaters or by more sophisticated gateway processors which are able to make routing decisions.

It is necessary to transmit control information as part of each data packet on a local area network. This control information indicates the source and destination of each packet, and allows a series of packets to be re-assembled into separate streams of user data. The U.S. Department of Defense's TCP/IP protocols are one such family of control structures which are publically specified (unlike some proprietary protocols) and are in widespread use.
Individual local area networks may be joined together using fibre-optic connections, satellite links or wide-area networks provided by the national communications authorities. The global Internet is the ultimate conglomeration of all these sub-networks into one huge infrastructure, and runs the TCP/IP protocols throughout. My networked transputers are directly accessible from anywhere on the Internet, subject to the usual access and security concerns. This provides similar access to transputers as is currently enjoyed by other architectures, and allows experimental hardware to be operated remotely.

1.5. Structure of this Thesis

Following the introduction, the remaining chapters of this thesis are structured as follows:

Chapter 2 surveys published literature in the fields of transputer hardware and software design methods, local area networks and performance enhancement techniques.

Chapter 3 briefly outlines the objectives of the research.

Chapter 4 looks at the detailed design of TCP/IP protocol processing software for the transputer, justifying the introduction of parallelism and the decomposition methods chosen. It concludes with evaluations of the software's throughput when measured on a number of single and multi-transputer hardware configurations.

Chapter 5 examines one application for this software - a networked transputer facility which provides access to transputers via a local area network.

Chapter 6 discusses several items of transputer-based hardware which have been developed to support this research.

Chapter 7 presents a second major application of the TCP/IP software - an embedded system containing transputers which pre-processes scientific data in real-time and transmits it to UNIX workstations for interactive analysis.

Chapter 8 summarises continuing activities in this research field, and considers how the design presented in this thesis relates to the new INMOS T9000 transputer architecture.

Chapter 9 concludes with a summary of the work done.
CHAPTER 2

2. Background and Literature Survey

The research builds upon many computing topics, some classical (such as data structures and performance-enhancement techniques) and some more recent (such as parallelism, transputers and local area networks). These have all been surveyed and the relevant background information is summarised here.

2.1. The INMOS Transputer

The INMOS transputer [INMOS89a] is a modern microprocessor which integrates a fast integer processor, a flexible memory interface, and an optional floating point processor with four serial communication links, enabling it to communicate with other transputer-family components in byte-synchronous fashion. The instruction set is particularly compact, and has instructions for synchronised communication built in. The processor contains a hardware scheduler, allowing it to time-share between a number of parallel threads of execution with short context switch times and very low overheads. A block diagram of the IMS T805 transputer is provided in figure 2.

Several features of the transputer greatly affect its performance, and had a significant impact on my code design and subsequent measurements. Firstly, all transputers have fast on-chip memory, usually 4 kbytes, and this is mapped into the lower part of the address space. The access time of on-chip memory is one processor cycle (40-50 ns, depending on the processor) whilst ordinary off-chip memory requires between three and five cycles per access, determined by the external address decoding and buffering circuitry. Unlike a cache, the effect of the on-chip memory is predictable, because the locations of its contents are always known to the programmer. The occam compilers attempt to place local variables in the on-chip memory, and then use any spare capacity for program code, starting with frequently-used library routines. It is important to note that small benchmark programs will make a proportionately better use of this memory, and therefore can execute significantly faster than if their contents were running as part of a larger group of processes on a single transputer.

A transputer’s serial links are driven by autonomous direct memory access (DMA) controllers. Once a communication instruction has loaded the relevant link’s DMA controller with details of the message to be communicated, it suspends awaiting completion of the activity. The transputer’s scheduler may then start another parallel process which is waiting to run (if there is one) whilst the communication is in progress. When the communication is complete, the original process is added to the queue of processes eligible for execution. During communication along a transputer link, the
data transmitted must be read from, or written to, the transputer's memory system. On a 32-bit processor, using a 20 Mbit/sec link, this activity will sustain 1.74 Mbytes/sec of traffic for unidirectional transfers, and 2.35 Mbytes/sec for bidirectional transfers. This corresponds to just 2% and 3%, respectively, of the on-chip memory bandwidth of a 20 MHz T800 transputer, but around 11% and 15%, respectively, of that of slower (5-cycle) external memories. These memory bus occupancies were considered when designing the memory access patterns of the individual protocol-processing algorithms - see chapter 4.

![Figure 2 - block diagram of a single INMOS transputer.](image)

[Peel92a] shows the effect of another form of DMA activity on the processor - that of servicing a slow peripheral device such as the AMD LANCE Ethernet controller, which occupies the transputer's memory bus for long periods during network packet transfers.

The INMOS technical notes [INMOS89b] and [Hinton93] are useful references on hardware design, although the latter was unfortunately not available when I was designing my circuitry.

Although the research reported herein has been designed for the first-generation 32-bit transputer family, INMOS (and latterly SGS-Thomson Microelectronics) have been working on a second-generation product, the T9000 family [INMOS91], [INMOS93a], [INMOS93b]. The T9000
processor shares the basic communicating-process philosophy of the earlier processors, but extends the data bus to 64-bits wide and adds a novel instruction grouping and pipeline structure to the processor. It also adds a virtual channel processor which, in conjunction with a new C104 routing chip, allows messages to be routed from any process in a parallel program to any other, regardless of how many intervening processors or routing devices they have to pass through. Another novel feature is the shared channel mechanism, which provides multiple client processes with a far more efficient mechanism to interact with a server process than does the basic alternation of the earlier products. Section 8.2 discusses how these new features could be used in a networked transputer device.

2.2. The Occam Programming Language

The occam programming language was designed at around the same time as the transputer hardware architecture, and the two operate very efficiently together. Occam is a language which incorporates parallelism, interprocess communication, timers and a channel selection mechanism (alternation). It implements the communicating process model pioneered by Hoare's Communicating Sequential Processes algebra (CSP) [Hoare85].

Occam is a block-structured language, and maintains scoped variables in the conventional fashion. Unusually, delineation of processes (blocks in other languages) is made by indentation of the source text; this aids visualisation and provides a standard look to the language, but ideally requires a folding editor to maintain the program's structure.

The communicating process model views a task as a set of parallel processes. Each element of a parallel process may run both sequential and further parallel code, and may harness the usual range of conditional, looping and procedure-calling constructs.

When parallel processes wish to communicate - a necessity if they are to perform co-operatively - one executes an input instruction and the other an output instruction, both referring to the same named communication channel. The first communication instruction to be executed causes its process to block until the second communication instruction is executed. Only then is the data transferred between processes, and only when that completes do both communication instructions terminate. A communication therefore acts as a synchronising mechanism between the processes.

In addition to simple channel communications, occam provides an alternation construct. This lists a number of input instructions - its input guards - and an associated list of processes - its guarded processes. When the ALT is executed, it checks each of the channels specified in its input guards to see if one or more is ready to communicate. If none is ready, it blocks until a channel tries to deliver a message, then runs the associated input process and its guarded process, and then terminates. If just one input guard is ready upon entry, it is selected immediately and the ALT proceeds as above. If more than one input guards are ready when the ALT is entered, an arbitrary
choice is made between them, and precisely one input instruction and its guarded process is run. A
prioritised ALT (PRI ALT) chooses the highest ready input channel in its list, rather than making
an arbitrary choice amongst its ready channels, which provides a different behaviour from the
ordinary ALT. (see section 2.3.4).

Timers are incorporated in occam as channels. A process may input from a timer whenever it
chooses, and the current (unsigned) value of the timer register is returned immediately. The
INMOS transputer increments its timer every 64 µs within a low-priority process, and every 1 µs in
a high priority process. It is also possible to request that the timer input instruction should block
until a specified time occurs. On the transputer, this delay mechanism is implemented by removing
the current process from the run queue, thereby allowing other processes to continue and the
delayed process to consume no resources whilst it is waiting.

A serious drawback with executing parallel processes on a single processor is that each process
might like to expand and contract its execution stack in a dynamic fashion in some circumstances.
This is difficult for a processor - such as the transputer - which does not have full virtual memory
facilities. Current occam implementations avoid the issue by banning recursion, and making all
declarations statically-dimensioned (including the number of elements in a parallel process).
Likewise, the occam specification does not allow functions to be passed as parameters to other
processes or functions. Thus the maximum workspace of each process may be calculated at
compilation time, and all such workspaces can be fitted into memory in a static manner. This
feature has a major impact on the type of TCP/IP implementation which can be produced for a
transputer, as discussed in chapter 4.

Occam takes extreme care to avoid the problems of accessing memory locations through more than
one identifier name in any particular scope (aliasing) and of accessing memory locations from more
than one element of a parallel process (parallel usage errors).

Aliasing gives rise to the possibility of side-effects, since we can update the memory location using
one name and fail to detect the effect on the rest of the algorithm which refers to it using another
name [Welch92]. A common scenario follows variable re-naming inside procedures, when the re-
named variable is also referred to by its previous, global, identifier. Occam outlaws such practices,
and aliasing is rejected by the compiler.

Parallel usage errors can give rise to non-deterministic results. Consider two parallel processes
which access the same memory location. If both of these accesses are reads, they will both return
the same result and the program will behave as expected. If one access is a write and the other is a
read, the result of the read will be determined by whether it is scheduled to execute before, or after,
the write. The value resulting from two parallel writes will similarly depend on which is scheduled
first. Banning such parallel accesses avoids the issue, allows the code to be partitioned across
multiple processors (since shared memory is essentially forbidden as a means of communication),
and allows programs to be checked for validity at compile time. Notice that aliasing must be
eliminated before the compiler can check for usage errors - it must know that each variable in scope in the various parallel processes refers to a unique memory area.

One final benefit of the formality of the occam language is that it is possible to draw up algebraic laws by which program fragments can be equated [Roscoe86]. This allows programs to be transformed from one form to another automatically, or (more usually), for two manually-converted versions to be proven to have identical behaviour. Recently, Goldsmith et al. have developed a full denotational semantics for occam2 [Goldsmith93] & [Goldsmith94]. These give a rigorous meaning to occam expressions, processes, declarations and abbreviations, and underpin current work in VLSI circuit design, the development of further occam program transformation tools and CSP to occam translations, amongst others.

The original version of occam (occam1, [INMOS84]) kept things simple by being typeless - all variables and constants were default, machine-length, integers. This permitted attention to be concentrated on the novel aspects of communication and parallelism in the language. In addition, occam1 had no functions and different syntax for procedure definitions, timer operations and array slicing.

Subsequent research led, in 1988, to the publication of the occam2 language design [INMOS88], which included functions, as well as extra non-integer data types (BYTES, BOOLs, REALs and different-length INTs) and methods for communicating such objects reliably between processes - channel protocols. The channel protocols are checked so that a value which is transmitted along a channel from one parallel process must be received into a variable of the same type upon arrival - essential to maintain security of the static memory allocation.

An occam2 function is rather like a procedure in that it is a named process and it takes arguments. Functions return results, whereas procedures return their results through variables in their parameter lists. The occam2 function may return more than one scalar quantity in its result, unlike most other languages. In addition, the use of a function has several other constraints; the parameter list may only contain constant (VAL) parameters - no channel or variable names; the function may not make use of or change any free (more-global) variables and the function may not communicate. These provisions ensure that the function always returns the same result every time it is run with a particular set of parameters; again, this purity of function is vital to reliable programming.

Occam2 is the version of the language which is in almost universal use today, including in this project. It only provides rudimentary libraries of basic arithmetic and input/output routines, and allows the user's separately-compiled routines to be packaged similarly. The structure is insufficiently rich to allow, for instance, variable numbers of parallel processes in such a library routine; this makes it impossible to supply some properly-buffered processes (such as my TCP router processes) in library-only form.
Since 1991, there has been considerable debate over the features proposed for a forthcoming occam3 - mainly to do with how to implement modular, separately compiled, processes in a semi-object-oriented fashion. The current draft specification for occam3 is [Barrett92]. About the only large-scale examples of occam3 programs are presented in [Kerridge93] and [Kerridge94a], although even these describe paper exercises since no occam3 compilers exist yet.

The most pressing requirements for occam3 compilers are the language features needed to control the new communication facilities of the T9000 transputer in a secure fashion. The main mechanism would be provided by a **SHARED** channel declaration, mapping onto the T9000's shared resource channels, plus the **CLAIM** processes and **GRANT** guards which are used by the client and server processes, respectively, to access it.

In the meantime, a subset of the proposed new occam3 features is being implemented into occam2.5 [INMOS94]. These provide minor enhancements to occam2, primarily the addition of named types and records, the ability to transmit record structures along channels, BYTE arithmetic, and several new operators relating to memory allocation, as well as a relaxation of the syntax in segmentation and in literal specifications. None of these radically alters occam2, and their inclusion would have no significant effect on the software reported in this thesis, except perhaps to the organisation of the protocols carried on the channels around the TCP root router process (see section 4.4.6).

### 2.3. Occam Idioms and Methods

Several years of experience of using the occam language have resulted in the establishment of a number of important programming paradigms (or idioms) in the same way that sequential programmers have developed many classical techniques for such tasks as sorting, linked lists or double buffering.

#### 2.3.1. Classes of parallelism - Geometric, Algorithmic and Process Farming

Thoroughly analysed in [Pritchard87], geometric and algorithmic parallelism and process farming provide classifications into which all parallel decompositions appear to fall. The goal of a good task decomposition is to minimise the overall execution time. This is usually achieved by ensuring that all the processors in a given configuration are kept equally busy, although communication times and synchronisation delays must also be taken into account.

*Geometric parallelism* is obtained when the data relevant to a task is sub-divided into sections, and each section is allocated to a separate processor - see figure 3. Processing will be efficient if the algorithm which processes each section of data can run unhindered by the need to exchange data with the other processors. Inefficiencies occur if data around the periphery of each section is also required by the algorithms running on neighbouring processors. Load balancing involves sub-
dividing the data space into sections with similar processing requirements. Many tasks suitable for processing using geometric parallelism are iterative and will loop repeatedly, so the processing requirements of each section may change with time and the load balancing may suffer because the sub-division is static (i.e. performed just once). The effects of these changes may be mitigated by striping or interleaving techniques. More random data allocation to processors can be even better, but is likely to increase the communication requirements excessively unless the processor communication hardware is particularly sophisticated. Geometric parallelism is inappropriate if the overall data space cannot be sub-divided into suitable sections.

Algorithmic parallelism can be utilised if the overall algorithm can itself be divided, irrespective of the division of data, so that the results of processing the data in one section can then be fed to others, and so on. This is most commonly seen in pipelined configurations (e.g. figure 4). The technique is only applicable if repeated computations can be made to flow continuously along the pipeline. Load balancing is successful if a division of the algorithm can be found where all the processes execute at approximately the same rate, minimising critical elements which restrict throughput. This may be achieved by choosing a processor of a different power for each stage, or by careful tuning of the algorithm. One important feature of a pipelined system is that the time taken to process each set of input values will be no less than if the whole task was executed on a single processor by itself. A second consequence is that each result will be generated at the end of the pipeline at the rate of processing of the slowest intermediate stage.

Figure 3 - geometric parallelism splits an array across nine processors

Algorithmic parallelism can be utilised if the overall algorithm can itself be divided, irrespective of the division of data, so that the results of processing the data in one section can then be fed to others, and so on. This is most commonly seen in pipelined configurations (e.g. figure 4). The technique is only applicable if repeated computations can be made to flow continuously along the pipeline. Load balancing is successful if a division of the algorithm can be found where all the processes execute at approximately the same rate, minimising critical elements which restrict throughput. This may be achieved by choosing a processor of a different power for each stage, or by careful tuning of the algorithm. One important feature of a pipelined system is that the time taken to process each set of input values will be no less than if the whole task was executed on a single processor by itself. A second consequence is that each result will be generated at the end of the pipeline at the rate of processing of the slowest intermediate stage.
It is possible to share the workload in a particular pipeline stage between two or more processes, on an alternating or an on-demand basis, to improve the load balancing. Obviously, this technique will be less beneficial if each sub-task takes a different time to be processed in the split stage, and results are required in the same order that the initial data was presented.

*Figure 4 - algorithmic parallelism*

**Process farming** is applicable when a task may be split into separate sections, with each totally independent of all the others. In this circumstance, a central farmer process may distribute the workload - comprising all the independent sub-tasks - to a pool of worker processes on a dynamic basis. This automatically provides good load balancing. The farmer, or a separate gatherer process, must collect the results of each worker's efforts when they are complete, so that the workers may continue with their next sub-tasks. On transputers, it is important to run the farmer-to-worker communications in parallel with the worker computations, because otherwise the worker processors will alternate between processing and communication, and the benefits of the DMA-controlled link hardware will be lost. This parallelism is usually achievable provided that the amount of computation per element of communication is sufficiently high. Figure 5 below shows two forms of processor farm. The first is a conventional scheme in which the controller recovers all the independent results and uses their receipt as part of the flow-control mechanism for generating new work requests. The second illustration (my minor enhancement of the principle in [Sturrock91]) shows how a process farm, incorporating any number of workers, may be incorporated into a pipelined process structure.

In each form of parallel task decomposition, there will be an overhead associated with communicating data between the processes, of starting the scheme (since there will be dependencies on previous results in most cases), and of bringing the scheme to an orderly halt with all work completed (since some processes will finish earlier than others). Only by exploring the load balancing and end-effect behaviours can the correct method of parallelism be chosen.
2.3.2. Deadlock Freedom

One of the most instructive works on deadlock is [Roscoe87]. This describes how deadlock can occur, mainly in the context of routing messages through a multiply-connected network of parallel processes. It starts off by describing a number of simple scenarios where deadlock does appear, and then goes on to analyse deadlock in rings of processes. Such a ring will be in deadlock if it contains one or more cycles of ungranted communication requests (i.e. commitments to communicate, each of which is blocked by the next). Likewise, a ring will be deadlock-free if at least one process in the ring is always prepared to accept input from its predecessor. In order to distribute this condition, each node must implement a buffer, and its ring process must only accept messages from its user process when the new message will not cause the buffer to be filled. Incoming messages from the ring may cause the buffer to fill up - but since this empties a slot in the buffer of the sending node, the ring will never become fully-committed with unsatisfied communication requests.
The paper then extends these rings, by adding cross-links, to derive more complicated structures, whose susceptibility to deadlock is tracked. Since it is possible to project a ring, or Hamiltonian Cycle, onto many common topologies [Nicole88], and to adapt any topology to one where Hamiltonian cycles may be accommodated by adding extra virtual channels, then communication schemes for all topologies may be developed. Extra cross-links may be added to reduce the diameter of the networks, subject to conditions which maintain deadlock-freedom.

This strategy can be applied to the pipeline structures produced in my research - deadlock can only occur if the cross-links between the transmit and receive pipelines are allowed to over-commit themselves, because network transmission always will be successful (although not necessarily immediately), and this breaks the main end-to-end cycle. Further details on deadlock may be found in section 4.1.10.

Shumway takes a rather different approach to avoiding deadlock when routing messages through regular mesh and hypercube structures in [Shumway89]. He divides the routing in such structures into a number of sequential operations across individual dimensions, and constrains all routing in the higher-order dimensions to be completed before work begins in the lower-order dimensions - an ordered dimension approach. Thus, cycles of dependencies cannot occur due to packets being routed in one dimension, in another and then back again in the first. Deadlock freedom for routing in each dimension is obtained by mapping a virtual ring onto each linear path, supplying sufficient buffers and using arguments similar to those in [Roscoe87] to justify the resulting behaviour. Many of the concepts in this paper are derived from [Dally87].

In the context of irregularly-connected communicating processes, the paper by Welch et al. [Welch93] is particularly useful. This describes two simple process paradigms, the client-server model and the I/O PAR model, and describes the restrictions which must be placed on their interconnection if the whole system is to be deadlock-free. A collection of I/O PAR processes communicate (once, in parallel) along each of their interconnecting channels, and then all go away to perform individual computations before re-synchronising to perform another parallel communication, and so on. This form of parallelism is easily shown to be free from deadlock. It is a common model for use in heavily-computational parts of a program.

![Client-server process model](image-url)
Two processes constructed in a client-server manner (see figure 6) are connected by two channels, a request channel from the client to the server, and a response channel from the server to the client. The latter might not be present in all circumstances. The client is always defined to start an interaction between it and its server. This is usually done by sending a client-to-server message, but can also be done by the client accepting a message in the reverse direction. The server must accept the initial message in finite time - just how long characterises the real-time behaviour of the combined system. After the first message, the client and server may exchange zero, one or more messages on the channels between them, in a pre-defined and therefore deadlock-free manner (although the precise sequence may be data-dependent). During a client-server interaction, the client may not perform communications outside of its sequence with the server, although it may continue to perform computations (provided that they terminate!). After the interaction has come to an end, only the client may establish another communication with this server.

The server is allowed to communicate outside of the process pair during a client-server interaction, with certain restrictions. It may therefore interrogate another server in response to an incoming request from its client. The external communications may only be performed as a client in another client-server relationship. By induction, it is not permissible, therefore, to have a cycle of servers cross-mounted and placing requests on each other - this can deadlock.

Figure 7 - overwriting buffer and prompter processes

Notice that a pipeline of processes may be modelled in the client-server scheme. Provided that the sink process at the end of the pipeline is always prepared to accept results in a finite time, each of the processes in the pipeline can be considered to act as a client to the process further along and as a server to the process further back, and the whole set remains deadlock-free. Notice, also, that the overwriting buffer (one part of figure 7) acts just as a server to two incoming channels, and places no client requests to any other process. An overwriting buffer and prompter placed in a cycle of channels therefore breaks the cycle of committed communications (going back to the ideas in [Roscoe87]), and also can be used in a cycle of cascaded servers to ensure that the servers cannot place requests on each other in a cyclic manner. Section 4.1.10 shows how this concept is used to prove that my TCP/IP processing pipelines are deadlock-free.
2.3.3. Multiple Buffering

I first presented a number of possible shared-memory buffering schemes in my paper at Enschede [Peel89]. The intention was to enable communication of data on either side of a pipelined processing task to proceed in parallel with the processing itself. In the case of a transputer-based pipeline, the ideal situation is that the Direct Memory Access feature of the transputer's links should be able to operate with virtually no hindrance to the processing task, or vice-versa. Discussion of the properties of multiple buffers was conducted by electronic means, and eventually published by Geraint Jones in [Jones89b].

The problem with multiple buffering involves the shared memory areas which must be accessed in parallel by the input communication process, by the processing element and by the output communication process. Occam's parallel usage checking rules do not allow more than one element of a parallel process to write to any memory location at once, or for one or more elements to read any memory location whilst another process is writing to it.

Jones proved which double-buffer constructions could be checked for parallel disjointness of their buffer elements, and which could not. In addition, he discussed triple-buffer schemes. The major pipeline processes in my code currently are surrounded by triple buffering, which cannot be checked by the compiler, but Jones' paper shows that the techniques used do not infringe the rules of parallel variable usage and are therefore reliable.

It is interesting to note that a true triple buffer (a computational process surrounded by two I/O processes, all running in parallel), just one of which is dealing with any buffered data at any time (see Example 1 in figure 8 below), may not be written whilst staying within occam's parallel usage checking rules. In this case, permission to reference each buffer element is passed around the chain of channels c12, c23 and c31. Although it is quite easy to justify that the values passing around this chain are unique, the compiler cannot check this statically, and is thus unable to determine whether, at any particular time, the buffer memories are being used by each of the processes independently.

One of Jones' alternative arrangements (see Example 2 in figure 8) can be usage checked, but has a behaviour which does not allow it to empty completely once it has been started - not much use for a pipeline carrying communications data!

Peter Welch, in [Welch91], provides a framework for managing pointers which provide access to shared memory buffers, and specifies the rules for their secure use.
2.3.4. Fair Selection

The occam definition of alternation specifies that an ALT process makes an arbitrary choice of which channel to receive from, should it be started with more than one of its input channels already prepared to communicate. Therefore, an ALT process could always choose to input from the same channel, when communication is pending on this channel (plus others) each time that the ALT is repeatedly invoked. This can lead to some undesirable behaviours, such as the one described in the next paragraph.

Consider a multiplexer process which is expected to merge the diagnostic output streams from two processes and to pass their results to a display device. If one process transmits from within a very tight loop, and its output is always chosen (arbitrarily) in preference to that of the other process due to the nature of the ALT process, then the diagnostics will appear out of the time sequence in which they were generated by the two processes. Also, the unselected stream will block its generating process and the effect will be for the diagnostics multiplexer to change the temporal behaviour of the whole application. A similar effect might be seen in my TCP multiplexer processes, where the
retry strategies of the transport-layer processes will become disrupted and the cyclic nature of the channels could even lead to livelock if the diagnostic output channels were to become delayed indefinitely.

The need for a fair, or at least a more deterministic, alternation mechanism was recognised by the community around 1988, especially in the work at Oxford on deadlock freedom. It was refined by several of us using the electronic media and was eventually published by Geraint Jones as [Jones89a]. Jones’ paper illustrates the problem, and shows several algorithms which ensure that all channels are serviced fairly. Some schemes are rather processor-intensive, and some can be implemented quite cheaply. They all rely on occam’s prioritised alternation (PRI ALT) mechanism, and re-assigning the priorities each time that it is invoked.

\[
\text{Example 1 - indices computed dynamically}
\]

\[
\begin{align*}
\text{INT} & \text{ favourite :} \\
\text{SEQ} & \text{ favourite := 0} \\
\text{WHILE busy} & \\
\text{SEQ} & \text{ PRI ALT index = favourite FOR n} \\
& \text{ VAL shifted IS (index REM n) :} \\
& \text{ c[shifted] ? x[shifted]} \\
& \text{ P(shifted)} \\
& \text{ favourite := (favourite + 1) REM n}
\end{align*}
\]

\[
\text{Example 2 - precomputed channel indices}
\]

\[
\begin{align*}
\text{[n * 2] INT} & \text{ perm :} \\
\text{INT} & \text{ favourite :} \\
\text{SEQ} & \text{ i = 0 FOR n} \\
\text{SEQ} & \text{ perm[i] := i} \\
& \text{ perm[i + n] := i} \\
& \text{ favourite := 0} \\
\text{WHILE busy} & \\
\text{SEQ} & \text{ PRI ALT index = favourite FOR n} \\
& \text{ VAL shifted IS perm[index] :} \\
& \text{ c[shifted] ? x[shifted]} \\
& \text{ P(shifted)} \\
& \text{ favourite := perm[favourite + 1]}
\end{align*}
\]

Figure 9 - occam code for two fair alternation implementations

Example 1 in figure 9 is perhaps the most direct method of determining each channel, in priority order, within the PRI ALT process. Unfortunately, this generates \((n+1)\) REM operations for each invocation of the PRI ALT, at a total cost of more than \(40n\) transputer processor cycles. This is clearly unacceptable. My enhancement of this algorithm, shown in example 2 of figure 9 pre-computes and stores references to all the channels and runs considerably faster. Jones presents several other schemes and discusses several priority-rotation schemes which deliver varying degrees of fairness.

2.4. The TCP/IP Family of Protocols

The mechanism for connecting computer applications together across a communication medium is best modelled by a number of distinct layers of software. These are required to control the network hardware, to make routing decisions, to perform error detection & correction and to interface the data stream to the user’s application code. The most common representation of this layering is
embodied in the International Standards Organisation’s Open Systems Interconnection (OSI) seven-layer model [Zimmermann80][Tanenbaum88].

2.4.1. The OSI Model

Figure 10 shows the basic structure of two interconnected hosts, each supporting the full seven layers of the model, plus an intermediate gateway which only requires the services of the lowest three layers.

![OSI Model Diagram]

The most important feature of the OSI model is that each layer behaves in a peer-to-peer fashion with its counterpart at the other end of a connection, although lower layers are actually involved in message propagation. The purposes of the layers are:

- Physical Layer: This comprises the physical communication medium, such as Ethernet cabling, optical fibre or radio signals. The Physical Layer specification defines the modulation characteristics and the bit rate.

- Data Link Layer: The data link layer provides a basic packet transmission service, in terms of the low-level packet structure, the addressing of physical layer devices, contention resolution on the physical medium, the retry strategy and so on.

- Network Layer: Since an OSI network may be built up from a number of interconnected subnetworks, one of the functions of software in each host is to choose where to send each packet. This is the responsibility of the network layer routing algorithm, which resides in gateway nodes as well as in the source and destination hosts.
Transport Layer

The three layers listed above provide a packet delivery service from any host on the network to any other, but delivery is not necessarily reliable because these layers might lose packets, duplicate them or re-order them by passing packets between intermediate nodes using different routes. The transport layer builds upon this unreliable service to synthesise a reliable one, providing flow controls and a stream-oriented view of active connections between source nodes and their destinations.

Session Layer

The session layer provides the user interface to control the streams of network traffic. It is involved primarily with supplying the information required to open connections and to close them again, but it also handles the negotiation of operating characteristics and provides a mechanism for reporting errors.

Presentation Layer

Some activities are performed on many OSI connections, such as character set conversion, encryption and bandwidth-saving compression of common forms of data. These can be performed at the presentation layer, to save having to incorporate them into the user's application program.

Application Layer

Finally comes the user's application itself. The underlying six layers of processing provide it with a standardised way of communicating, which will not change regardless of distance, the number of intermediate sub-networks traversed, the media used, or the reliability of those media.

Many functions of the OSI model have been undergoing detailed modification until relatively recently, and consequently there were few full implementations in common use when this research was started. The OSI protocols are still not in widespread use on the University of Surrey campus, although they are gradually gaining commercial and academic acceptance elsewhere. For these reasons, the OSI protocols were not chosen for this study; instead, the similar Internet (TCP/IP) protocols are used.

Almost all the machines on the Departmental Computer Network currently exchange information using the Transmission Control Protocol and Internet Protocol (TCP/IP) family of networking software. This protocol combines the six functional layers of the OSI model into just three layers beneath the user's applications, which are pictured in figure 11. The boundaries between these three layers do not precisely match the boundaries of the OSI model, but the principles are the same.
2.4.2. The TCP/IP Protocols

TCP/IP was designed for the US Defense Advanced Research Projects Agency (DARPA) as a protocol family to support its military and defence-research computer network, ARPANET. All research on the interconnection of local area networks was done with reference to a global network, however, which took the familiar name ‘Internet’.

TCP/IP defines the addressing scheme for all host computers connected to the Internet. These are known as Internet addresses, and are 32 bits long. Since so many computers are now being connected to the Internet, and since numbers are allocated in groups (e.g. the University of Surrey has a group of 65536 addresses allocated to it, although only about 1400 of these are in current use), the Internet address space is close to filling up. In just a few years, alternative numbering schemes will have to be introduced. [Bradner95 - RFC1752] contrasts the best alternative proposals and recommends a new IP-layer addressing scheme which should prove adequate for the foreseeable future - it will use 128-bit addresses. In contrast, [Onions - RFC1606] (in jest? - note the date) conjectures upon the need to be able to address every light switch and mains socket, in addition to body-monitoring sensors which may be injected into the bloodstream, and individually-addressable consumer packaging (for the market researchers!). Let’s hope that many more aspects of security and reliability will have been solved by then ...

There are many dozens of individual protocols in the TCP/IP set, some of which were originally experimental and are now rarely used. See [Comer91a] and [various RFCs] for details.

![Figure 11 - the TCP/IP protocol structure](image-url)

The major protocols which are central to all TCP/IP installations are :

- **IP** The Internet Protocol [RFC791 - Postel81a] is the lowest layer of protocol processing above the hardware layers. IP is the TCP/IP equivalent of the network layer in the ISO model. It manages the routing of messages which arrive from other hosts or from higher up its own host’s protocol stack. It holds a table of hardware addresses of other networked
hosts that it is directly connected to, and chooses one of these as the destination of each packet that it is asked to transmit.

The IP header which is added to every packet which flows across the network specifies, amongst other things, the Internet addresses of the source and destination hosts and a unique serial number which allows identification of the packet if it should become duplicated. IP is also responsible for sub-dividing, or fragmenting, packets which are too long for particular physical network architectures, and for re-assembling the fragments for each IP packet when they are received.

**ICMP** The Internet Control Message Protocol [RFC792 - Postel81b] specifies a number of responses which may be generated asynchronously by hosts on the Internet, and sent to other hosts to let them know about congestion, routing problems, and other such low-level functions. Within a protocol stack, ICMP messages must be copied to whichever processes have expressed an interest in that type of message, often causing layering boundaries to be blurred.

**TCP** The Transmission Control Protocol [RFC793 - Postel81c] is responsible for maintaining error-corrected and flow-controlled streams of information between user processes. It performs the duties of the transport layer in the ISO stack. Because the ISO layers higher than the transport layer are not directly provided by TCP/IP, TCP itself must provide an interface to the user's process, and the user must be responsible for the ISO presentation layer activities.

TCP operates by splitting the bidirectional stream of data between two processes into separate packets. Each packet is transmitted complete with a comprehensive header which identifies the stream, specifies where in the stream this particular packet lies, and provides information about the 'window' of bytes in the stream which it is prepared to accept next. This *sliding window* therefore indicates which bytes have been accepted - thus acknowledging them to the transmitter - and also shows how many more bytes the receiver is currently able to accept, which restricts the flow of traffic to that of its buffering capacity [Tanenbaum88]. The transmitting host re-sends packets if it does not receive acknowledgement that they have been received within a dynamically-adjusted *retry interval*. Since each TCP packet contains information about its host's receive status, it is possible to send traffic in both directions without any extra acknowledgement packets. If a host has no data to send, it must transmit empty acknowledgement packets in response to incoming data packets, providing information about traffic flowing in the opposite direction.

**UDP** Unlike TCP, which is designed to be fully reliable, the User Datagram Protocol [Postel80 - RFC768] is an unreliable datagram protocol. This means that UDP packets are passed across the network without first establishing a one-to-one connection with a receiving
process, and without the protocol defining any acknowledgement mechanism. This accommodates broadcasting, but only of relatively short individual messages which fit, in their entirety, into a single network packet. When a host receives a UDP packet, it must duplicate it to any user processes which have shown interest in its particular port identifier number. Since such messages may arrive faster than the user processes can deal with them, only a queue of recently-arrived messages need be stored.

TFTP  The Trivial File Transfer Protocol [RFC783 - Sollins81] provides a simple mechanism for exchanging large amounts of information without going to the expense of using the more complicated TCP protocol. This is most useful for loading an application or an operating system into a disc-less workstation (or a stand-alone transputer) which only need store a small bootstrap loader in read-only memory. TFTP is a Positive Acknowledgement / Retransmission protocol [Tanenbaum88], meaning that each packet must be acknowledged before the next is sent. This is slow - the latency of such communications and the small (512-byte) maximum packet size restricts performance over Ethernet to about 200 kbytes/sec. If an acknowledgement for a packet is not received in a reasonable time (usually 1 second), the packet is re-transmitted.

ARP  Each host on a local area network is allocated a unique 32-bit Internet address. It also has a physical hardware address (48-bits long in the case of Ethernet), which is usually stored in hardware or read-only memory on each network interface card. When a new host is connected to a network, it is necessary for it to discover the physical addresses of any other hosts with which it may wish to communicate. The Address Resolution Protocol [RFC826 - Plummer82] is a simple datagram-based protocol which does this. When the new host wishes to talk to another host for the first time, it broadcasts a request for that host (specified by its Internet address) to respond. The outgoing message carries the hardware address of the requester, of course. A received response carries the hardware address of the other host, allowing the original communication to proceed, and the addressing table in the IP processes of both hosts to be updated for future use. It is usual for hosts to implement table ageing so that unused entries in the ARP table are deleted after a period of inactivity. This ensures that stations which change their addresses do not cause confusion, as well as keeping the tables small.

RARP  The Reverse Address Resolution Protocol [RFC903 - Finlayson84] enables a host, when it is first booted, to broadcast its own physical hardware address onto the network, and for another host to respond by returning its Internet address. This is useful, since it allows disc-less workstations to discover their identities from the central register before joining the network. After successfully receiving a RARP response, the workstation can then use its newly-discovered Internet address in TFTP exchanges, in order to load its own code before starting to run normally.
Telnet streams [RFC854 - Postel83] carry terminal traffic between host applications and display devices. Telnet uses the TCP protocol as the underlying transport mechanism, and adds two extra features - control code handling and an interrupt mechanism. Control codes are preceded by an escape character in order to insert them into the set of displayable characters, allowing the characteristics of a range of terminal types to be accommodated. Interrupts are handled using the TCP urgent pointer header field, which allows the host processor to discard incoming characters if a break-in is detected further along the stream. [RFC854] also specifies the user dialogue required to specify the characteristics of a connection when it is opened.

Others

There are several other TCP/IP protocols which are very common but which will not be discussed further here. Most are concerned with transmitting specific forms of user data, such as whole file transfers and terminal traffic.

FTP

File Transfer Protocol streams [RFC959 - Postel85] carry complete files according to commands entered by the user. The scheme uses an underlying TCP connection, and is therefore much faster that the TFTP protocol.

RIP

The Routing Information Protocol [RFC1058 - Hedrick88] is one of a number of mechanisms designed to allow hosts which connect individual local area networks to exchange information about the routing of packets. On each local network, gateway hosts occasionally broadcast the network numbers of other networks which they know about, providing knowledge of connectivity and avoiding problems when connections fail.

DNS

The Domain Name Service [RFC1034 - Mockapetris87] provides an IP address directory which prevents each host on the Internet from having to maintain a table of every other host's IP addresses. The directory is held hierarchically, so that subsidiary name-servers which cannot perform a translation themselves simply pass the request up a tree to a superior name server which can.

Two further specification documents have a significant influence on recent TCP/IP software designs - [RFC1122 - Braden89a] and [RFC1123 - Braden89b].

RFC1122 is a summary of all the network-layer and transport-layer protocols (including IP, TCP, ARP and UDP). It talks through many grey areas in the original specifications, including a few inaccuracies and a number of ambiguities. More importantly, it discusses the experiences of many implementors, especially regarding performance. Inclusion of its lessons is particularly important for TCP/IP implementations which will be connected to non-local networks.

RFC1123 provides a similar updating and bug-fixing service for the higher-level TCP/IP family protocols, especially Telnet and FTP. The contents are of lesser significance to the work reported in this thesis.
2.5. Transport Protocol Implementations on Sequential Machines

One of the earliest discussions of TCP/IP performance is [Clark82]. This was written when typical mainframes could execute only 2 to 5 million instructions per second, and minicomputers far fewer. Microcomputers had not progressed beyond the PC-XT level, and Ethernet cards probably were not yet available for them. Clark argued that Ethernet throughput - measured in terms of the packet transmission rate - was primarily dictated by the operating system’s scheduling overhead, which directly affected the maximum interrupt rate. Of course, this would only be true if each packet was used efficiently, and thus it was important to ensure that each data packet carried its maximum permitted allocation of data, and that as few small acknowledgement packets as necessary were sent. The other dominant factor in TCP processing was the per-byte overhead - copying data from one place to another and checksumming it.

Clark went on to look at where protocol code could be located - in a user-level process under control of the scheduler, in the operating system kernel itself or, completely separately, in a front-end processor. All three of these give rise to problems - especially now we are considering multiple-process, layered protocol implementations.

- There are advantages to developing protocol code in ordinary user processes (remember, many protocols were still being prototyped at that time), since it is less likely to interfere with other applications. On the other hand, calling user-level code from other application processes is awkward because the security boundaries between the kernel and all the higher-level processes are broken.

- Programming at the kernel level is difficult - and potentially damaging - but places the protocol code in its logical place. One concern with this is that all protocol processing will tend to be performed as part of interrupt service routines, and that further interrupts may be disabled for an unacceptably long time. Also, if the network is particularly busy, perhaps due to a malfunctioning host elsewhere, the kernel will monopolise the processor to the detriment of user processes. In 1982, there was also concern about the size of the kernel, which appears to be ignored today and spoils many common operating systems and software packages.

- Placing all the protocol processing code on a front-end processor is a good way of isolating the protocol code, and maybe makes it easier to write, but the cost (in terms of interrupts and data copying) of communicating between the front-end and main processors is similar to that of doing all the work on the latter. This communication is not dissimilar in complexity to the original TCP/IP protocol, either.

In all these schemes, implementing layered protocols in separate processes is a mixed blessing because the layering enforces large amounts of data copying between workspaces. It also forces work for different users to be brought together into routines which have to be able to handle connections with very different characteristics. This suggests that the problem of where to
implement the protocol code is one of data-stream multiplexing and of data delivery to the user process. Clark pointed out that bundling some of the protocol code - TCP for example - in with the application would be efficient, if difficult to write and to maintain. He even suggested that providing many different TCP processes, each tailored to different connection behaviours (such as high throughput or low latency, for example), would be worthwhile.

[Clark82] concluded with comments about particular problems of the two main TCP/IP protocols. IP code is mainly concerned with packet reassembly and addressing table lookups, and special care must be placed in the design of these algorithms. Cached lookups, placing strong reliance on the common case of packets belonging to the same connection as other recent ones, are an important optimisation in this instance. TCP processing is dominated by checksum calculation, header option parsing, and buffering and distribution of individual data streams to the user processes. It can be optimised by assuming that the current packet is the one expected - and that the common path through the code can be followed. Similarly, algorithms should de-emphasise rarer cases such as retransmission of unacknowledged data, since this only happens occasionally in practice. Data copying is expensive and should be avoided whenever possible, although the Berkeley BSD versions of Unix do provide a central buffer pool (of mbufs) which avoids this. Such shared memory techniques cannot sensibly be transferred to parallel implementations on communicating multiprocessors - see section 2.6 for further discussions of these issues.

Surprisingly little detailed analysis and measurement appears to have been done on existing protocol implementations across local area networks. Smith & Kain [Smith91] reported on measurements of a 33-station Ethernet, concentrating mainly on packet size distributions, distributions of packet sizes at various total Ethernet loadings, network collision rates and queueing delays at the Ethernet controller whilst packets were waiting for transmission onto the network. Many of these results were borne in mind whilst the code in this thesis was being designed. Particular note was taken of the relatively low average packet size recorded as the network became congested (about 200-250 bytes).

Heatley & Stokesberry reported a series of performance measurements in [Heatley89]. The hardware used for each station was an Intel 80286-10 processor, which generated the test data, front-ended by an Intel 80186-8 processor which ran the protocol-processing software and managed an Ethernet controller. The two processors were connected together using shared memory; each also had local memory for all other purposes. The report examined commercial ISO TP4 transport-level software running with both IP and null network-layer support. Its highest reported throughput, for an optimal choice of user message size, and with checksum calculation disabled, was about 230 kbytes/second. The effects of processor saturation and data copying overhead were clearly shown.
[Clark89] described work done to instrument TCP/IP code which started out as the Berkeley implementation for Unix but which was then highly streamlined to optimise its performance. In particular, they replaced all the Unix system calls with sophisticated stand-alone routines. The main results provided are instruction counts for each of the major packet transmission and reception stages - which they suggest do not vary much between their Intel 80386 and alternative Motorola 68020 or SPARC processors. Since the authors were amongst the main designers of the TCP/IP protocols, not surprisingly they conclude that the protocols themselves are not too expensive to run fast. Optimisations which deserved special attention included the prediction of TCP headers, buffering between the IP, TCP and user layers, timers and schedulers. The final receive rate which they predicted (of 25 000 packets/second, assuming 300 instructions per packet on an Intel 80386 processor at 10 MIPs) must be discounted by a third to account for acknowledgement packets which must be transmitted in the opposite direction. This figure does not include any data movement or checksum computation, but at least shows that TCP/IP should be able to saturate an Ethernet with 14 000 zero-length packets per second, which would normally be used to carry just acknowledgement information. Whether an 80386 can sustain 10 MIPs, and therefore whether this conclusion is valid, is debatable.

A second set of experiments in [Clark89] (using a 20 MHz Motorola 68020 in a Sun 3/60 which they rated at 2 MIPs) attempted to quantify the data copying and checksumming overhead. The checksumming could be performed (in memory) at 8 Mbytes/second. Copying of this data between buffers was performed at a similar rate, but copying to the LANCE Ethernet controller was only half as fast - they observed the same DMA overhead that I have reported in section 4.1.1.2 of this thesis. Extrapolating their figures, such a processor was capable of 825 maximum-length data packets per second, or around 700 per second if TCP acknowledgements are taken into account, giving a user-data transfer rate of fractionally over 1 Mbyte/second. This is just over twice the rate which I measured in early tests with Sun 3s, probably reflecting the extra costs incurred in calling the TCP code from the user level and passing in the user data, as well as those for extracting the data at the receive end of the connection. It also reflects their optimisations over the standard SunOS product.

These figures are useful confirmations of the transputer figures given later in this thesis - a 20 MHz transputer with 5-cycle memory being comparable in performance to 80386 and 68020 processors in many respects.

Weaver, in [Weaver91], discussed many of the pitfalls of existing transport protocols, as part of a justification for his new protocol, XTP. Those which are worth bearing in mind when re-implementing TCP/IP include:

* The size of the data pipeline is dependent on the amount of buffering between the producer and consumer applications, and this is limited in some protocols (such as TCP) by the maximum size of window which may be advertised.
• Parsing protocol control information is expensive, especially if packets contain a variable number of options which have to be extracted in sequence. The position in the packet of checksum fields can make a large difference to how they are handled in software, or in hardware.

• The complexity of setting-up and breaking down connections is particularly important if only a single data packet is sent during a connection. Typically, about six other control packets will be required to manage this single data packet, at a cost of high latency and poor network occupancy.

• Flow control must be regulated so as not to swamp the slowest part of any connection path. Most protocols, driven too fast and into packet loss, resort to re-transmissions and consequent waste of network bandwidth.

• Retransmission of the whole transmit window when a packet is not acknowledged is often unnecessary. If the receiver were able just to request that a small section be re-sent (by sending a negative acknowledgement), far less traffic would ensue. Likewise, if the transmitter was able to request acknowledgements, rather than expecting an acknowledgement response to virtually every packet that it sent, much network activity could be avoided.

• Data alignment is often required - for instance to make checksum computation less expensive - but network packets often do not carry the important fields at word boundaries. It may be possible to specify byte offsets in the packet so that such alignment does not need to be done when the packet is constructed. Notice that, on the transputer, a []BYTE channel communication performs automatic re-alignment to the destination memory location, and thus alignment comes free when data is moved between processes.

[Weaver91] concludes that the TCP/IP protocols are far from perfect for high-speed transmissions, and much less suitable for implementation in hardware than XTP. Many of his points are useful starting points for a distributed processing solution, however.

Several other authors have also proposed hardware assistance for some parts of the protocol processing workload. Siegel et al. proposed that a reduction of facilities in the transport protocol would be a good basis for a hybrid hardware/software implementation of the ISO TP4 protocol [Siegel91]. In particular, they relegated checksum calculation to special-purpose hardware. They also restricted the size of transport-layer packets to prevent further fragmentation in the lower link-level layer, and simplified the code for managing windows by avoiding some window resizing features of the original protocol, such as never closing the window to zero and thus having to detect when it re-opened. These strategies were then sub-divided and implemented on a multi-CPU front-end processor architecture which shared its memory with the host processor's data bus and with the physical layer medium-access hardware.
A particularly high-performance hardware-assisted front-end processor has also been implemented recently at Hewlett Packard [personal discussion]. This achieves 200 Mbits/second throughputs when used with their current range of high-performance RISC processors.

An alternative to using a heavy-weight transport layer protocol is to develop and use a simpler one, perhaps tailored to high-speed block data transfers. Common alternatives are XTP [Weaver91 - discussed above], HTPNET [Chan93 - see section 2.6], LNTP [Atkins88] and NETBLT [Clark87]. All of these attempt to streamline the header processing, in particular, by re-arranging the various fields and thus permitting more efficient use of pipelining. XTP, for instance, places the checksum at the end of the packet so that it may be inserted (by hardware) after the rest of the packet has already flowed past. HTPNET concentrates on reducing the overheads of error management and packet retransmissions, with a much simpler retry strategy and a reduced reliance on timers. LNTP reduces error control by relying on the Ethernet-level hardware checksum only. It also separates the two bidirectional halves of a connection so that data may not be piggy-backed onto acknowledgements but must be carried in separate packets. This allows received acknowledgements to be sent straight to the transmitter process, and decouples the transmitter and receiver processes - something which I found very difficult with TCP - see section 8.1. NETBLT is a transport protocol designed for bulk data transfer, and attempts to reduce overheads at the user-layer to transport-layer interface by passing large data buffers rather than simulating a continuous data stream. Each data buffer is actually transmitted as a train of maximum-sized network packets, and is only acknowledged once, at the end. If errors are present, just the missing packets are requested to be re-transmitted. Once the whole buffer has been successfully transferred, work starts on the next one. This lock-step scheme is very different from the other protocols and enhancements reported in this section. NETBLT also incorporates a flow-control scheme which works over a wide range of data paths from low-latency local area networks to very high-latency satellite links.

2.6. Parallel Implementation of the TCP/IP Protocols

On the transputer, some of the objections to processing each layer of a layered protocol in a separate process are invalid. The hardware scheduler on the transputer performs extremely fast context-switches, and executing the layer processes on separate transputers will involve passing the data between these machines using the DMA-controlled serial links provided on the processors; thus data copying is essential, and relatively slow, but it will be performed in parallel with other computation and therefore has less of an impact than might be expected.

A number of groups have investigated the use of parallel computers for protocol processing. Some have even used transputers. The summary in this section shows the areas which have been researched.
INMOS, with their IMS B300 network interface product, have produced a device which sits on an Ethernet and allows networked users to obtain access to transputers in a similar way to my networked Iserver which is reported in Chapter 5 of this thesis. The B300 hardware comprises a TRAM motherboard, populated with a range of standard transputer modules (or TRAMs). In total, it provides support for four subsidiary transputer networks as shown in figure 12.

![Figure 12 - the INMOS B300 - hardware architecture](image)

The device relies on an implementation of the TCP/IP protocols which are written in occam, but around an mbuf-style model, as alluded to in the TCP specification [RFC793 - Postel81c], and implemented using shared memory in the Berkeley BSD series of Unix operating systems. INMOS chose this architecture, but isolated the permissions to access the shared mbuf memory in a server process which runs in parallel with the rest of their processes. It is thus necessary to send a message to this process to ask for permission to work with a particular buffer, and to send another message to relinquish the permission before a pointer to the buffer is sent to another process. Without extra high-latency message routing code, this permissions structure restricts the design to running on a uniprocessor - just one T425 handling all the layers of the TCP/IP processing, for all open connections.

Together with Graeme Tozer (who wrote the INMOS software), I suspect that the performance of the INMOS software is limited by the expense of running the permissions process. Certainly, the maximum reported performance of 200-300 kbytes/sec which this code is able to sustain is a factor of three slower than my more parallel approach. Other reasons for the limited throughput include the transfer of data via the T222 to three ports - which run around 10% slower than the other one - and the high latency of the socket-based software interface.
Other intriguing features of this architecture are the VDU-based monitoring system (requiring extra RS232 conversion hardware, with limited throughput and persistence, and with only a printer port for recording purposes), and the decision to provide four subsidiary transputer networks, since one must be driven from a different transputer (the T425) from the other three. There is no detailed publication on this development, for commercial reasons.

Perihelion and Parsytec sell a version of TCP/IP which I believe to be a direct port of the C-language BSD sources. Again, this code will only run on a single transputer, and is also limited by the high latency ring architecture which the Helios operating system uses to distribute the TCP/IP traffic to the user processes.

Martina Zitterbart, Torsten Braun and colleagues at the University of Karlsruhe are researching into the use of transputers for implementation of a wide range of protocols, including XTP [Braun91b], the ISO stack [Braun91a] and [Zitterbart91] and TCP/IP [Koufopavlou92]. Zitterbart observes that the processor loading of the OSI TP4 state machine, or of TCP's, is only 20% to 30% of the total processing load. The rest, as shown in section 2.5, above, comes from the data copying, buffering and checksum elements. Rather than provide a hardware implementation of the 20% part, the Karlsruhe team have designed a general-purpose transputer-based scheme which introduces parallelism to this part, using a pipeline to process the various protocol layers, and employs global
memory processes to avoid passing data along the pipeline. The basic software architecture of all their implementations is shown in figure 14. Much of their work is concerned with automation of the coding, using tools which generate code for the transputer from the formal specifications of these protocols [Braun93]. Unfortunately, TCP/IP is far less easy to specify in this way, due to its less formal specification (and, indeed, specification errors!).

![Diagram of the Karlsruhe transputer-based protocol-processing architecture](image)

*Figure 14 - the Karlsruhe transputer-based protocol-processing architecture*

In each case, their papers report that the protocols have been exercised under control of their hybrid NETMON hardware/software simulator. They have obtained impressive performances from their software, but do benefit in some instances from choosing protocols which do not require checksum computations, and they do not appear to be passing the actual packet data to and from the global memory - just all the pointers necessary to simulate it. Their architecture, therefore, could only be used for real work if shared memory were provided between the processors. Results in [Zitterbart91] (for the ISO protocol) show about 500 packets/sec when all the processes shown in figure 14 were run on the same processor, about 700 packets/sec when the test, simulator and the two memory managers were run on four transputers and the protocol processes on a fifth, and about 2050 packets/sec when the protocol processing was sub-divided onto four transputers, making eight in all. The throughput corresponding to this last figure is quoted as 137 Mbit/second, which would be impossible to communicate via one, or both, of the memory manager processors using all their links. Interestingly, the software architecture reported in this thesis is capable of transmitting and receiving over 1070 data packets/second, when carrying bidirectional data between two 20 MHz T800 transputers, even when each packet contains 1400 bytes of user data - a throughput of 1.5 Mbytes/second. In terms of parallelism, it would correspond most directly to the 700
packets/sec figure above, but remember that the latter does not incorporate the data transfers. Neither figure includes checksumming - it being ignored in the Karlsruhe case and relegated to the TCP router process and run in parallel in my case. It might be necessary to double the 700 packet/sec figure, depending on whether the measurement counts the processing of each TCP packet at each end of each connection separately, or just once, but the lack of data bytes passing through the protocol-processing processors is significant regardless.

The performance figures reported in [Braun91b] are somewhat sketchy, but suggest that the time taken to process a single received XTP packet is 1900 µs and 1350 µs for a transmitted one, plus 25% in each case for the communication between the various processes. When joining two transputers, one sending and one receiving, this implies that 421 packets are transferred per second, which is not particularly impressive.

[Koufopavlou92] is a design study for a proposed TCP/IP implementation, rather than a report of a real implementation, like the other two papers above. It splits the complete TCP processing workload into about a dozen separate processes. These illustrate the numerous asynchronous activities which could be performed in parallel, provided that global, shared, access to the transmit and receive data buffers is available. It provides estimates as to the number of instructions each section might take, as in [Clark89], looks at the maximum parallelism available when processing each packet and arrives at per-packet times of 6.2 µs for sequential execution or 2.2 µs for fully-parallel execution. These figures ignore data movement, plus any overheads of process synchronisation, communication or sharing of the important state variables, and translate to 160 000 packets per second - very fast. With many of the same parameters, however, [Clark89] arrived at a figure of 40 µs for the sequential case, and the three-times speedup for the parallel case would involve some very fine-grained parallelism which could be expected to have substantial communication overheads if implemented on transputers. The paper concludes with very realistic remarks about the global data memory access rates and degree of multi-porting which this architecture would require, concluding that the memories would probably be the limiting factor to overall throughput.

[Braun91a] reports on the use of a similar architecture that simulates the processing of thirty 64 kbit/sec ISDN channels, and their gatewaying onto a local area network. The protocol processing for this routing application is all performed at the network layer of the ISO model. The simulation results state that the data content of each packet has been kept very small to minimise the traffic to and from the memory managers - presumably as was done in the experiments reported above. Using 4 transputers for the protocol processing, receive rates of 1200 packets/second and transmit figures of 1900 packets/sec are given. These rates do appear to show some sensitivity to the number of ISDN ports used - the expense of the addressing table look-ups showing in the way that most references suggest it should.
The Karlsruhe work is a significant effort which provides some interesting comparisons with the work reported in this thesis. It would be interesting to develop it further, to replicate the memory manager processes (using geometric parallelism), and therefore to provide sufficient link paths to carry the data rates which the protocol processors can handle. Provided that these memory managers could be accessed fast enough by the protocol processors not to become a bottleneck, and provided that load balancing across the memory managers could be achieved, then this architecture could provide an exceptional performance.

The IBM Zurich Research Laboratory's Parallel Protocol Engine (PPE) [Rutsche93] builds upon the Karlsruhe architecture to provide a front-end protocol processor for scientific workstations. The PPE uses four transputers for the protocol processing, and memory shared amongst the transputers, the workstation and the network interface hardware to store the user data - as pictured in figure 15.

![Figure 15 - the IBM Parallel Protocol Engine](image)

The Parallel Protocol Engine implements a shared-memory version of TCP which maintains its transmit and receive data in fast buffers. These buffers are referenced through data structures (the Task Control Blocks or TCBs) which also store the state variables for each connection. The TCBs are stored in the protocol processors and are accessed by the other protocol processors using transputer channel communication and a virtual shared memory organisation. [Rutsche93] provides performance figures for two scenarios - one where real user data is copied from the
workstation through to the network and one where the data copies are substituted by null transfers. In both cases, TCP checksumming is not done, with the justification that it should be relegated to the network interface hardware. Two PPEs are connected back-to-back, with the two network interfaces directly connected. With data handling (scenario 1), a throughput of about 2000 packets/second is achieved for 1500-byte Ethernet-sized segments using the four-processor configuration pictured above; this increases somewhat for larger (FDDI-sized) segments and falls to about 1400 packets/second with just one processor. With data handling omitted (scenario 2), some 3400 packets/sec can be handled with four processors and 2500 packets/sec with just one processor. These relatively low speedups are justified by the poor load balancing (two processors are used for TCP and two for IP, and their loads are not equal) and by the overheads of the virtual shared memory access to the TCB data structures. The paper concludes by speculating on the performance of a re-implementation using two T9000 transputers.

[Maly93] proposes a process-farmed approach to TCP processing on a Sparc-10MP processor which has four SPARC processors and a global memory unit connected by an 800 Mbytes/sec high-bandwidth bus. The paper analyses these processors when running TCP/IP across 200 Mbit/sec FDDI connections, using the instructions-per-packet method of [Clark89], and presents simulations which show that speeds of some 170 Mbits/sec should be achievable. Although not very relevant to my first-generation transputer work, the paper sets some interesting goals for T9000 implementations which will be discussed in section 8.2.

Like the Karlsruhe and IBM activities, Gorton and Chan's work on HTPNET protocols and implementations at the University of New South Wales [Chan93] and [Chan94] are aimed towards producing hardware-assisted protocol handlers. Gorton & Chan have the additional goal of supporting the throughputs of future optical networks - up to 1 Gbit/second. At these speeds, the limitations of transport protocols such as TCP and TP4 become insurmountable, due to the expense of the retransmission schemes and the considerable amounts of window buffering required. This has led them to develop a new, simpler transport protocol called HTPNET. It is designed to operate in unidirectional mode over a connection-oriented broadband network (such as B-ISDN, or ATM) and therefore does not have to worry about addressing or the order of packet delivery. HTPNET provides separate packets for data and for control information, and provides a selective retransmission mechanism, by which transmitted synchronisation messages solicit a response which contains the sequence number of the most recent packet received, plus notification of any gaps in the received packet sequence. This scheme is more efficient than TCP's go-back-n retransmission scheme, in which all unacknowledged packets are sent again, provided that the error rate is low, and bursty [Tanenbaum88]. At higher error rates, HTPNET reverts to a retransmission strategy which closely mirrors go-back-n. The separate control strategy allows the protocol to be specified as a set of finite state machines, rather than just one, and this makes it more immediately amenable to parallel processing, or casting into silicon.
Gorton & Chan have obtained very high simulated performances from an architecture which uses shared memory to pass packets between transputers. In the earlier paper, the deficiencies of the buffer sharing were eliminated by simply ignoring the data transfers. Recently, they have started to use considerable amounts of external hardware to handle the data, although transputers are still used to manage the protocol. Remember, too, that HTPNET has been simplified to make this approach more suitable.

With a total of five T800 processors running the HTPNET protocol, throughputs of 13,000 data packets per second have been reported, but comparisons with TCP are difficult. HTPNET does not perform the network-layer activities, does not include control information in each data packet, and in these T800 measurements, there is no data management. Thus, the data packets which are generated are small and require little computation to produce, and this is all that the packet rate really measures.

[Chan94] describes a proposed HTPNET architecture built around T9000 transputers. In this, incoming network packets are streamed along a 32-bit hardware bus to one of a number of dual-ported video-RAM FIFOs. As each packet arrives, it is inspected by a T9000, whilst the neighbouring FIFOs continue to accept successive packets. The T9000s communicate to ensure that packet sequence numbers are contiguous and sequential. Correctly-ordered data is passed to the host processor’s memory along another 32-bit bus - this is a front-end processor configuration. The parallelisation of the data memories addresses the concerns about memory bandwidth mentioned in [Koufopavlou92]. The FIFO T9000s report packet progress to a single control T9000 which generates synchronisation packets and manages re-transmission. The architecture has been simulated at network speeds of 200 Mbit/sec and 1 Gbit/sec, from which conclusions about data packet size and numbers of data FIFOs have been derived.

Diot, Ng & Dang write about a proposed implementation of the ISO TP4 protocol in [Diot90]. Their architecture relies on shared and dual-ported memories to handle all the data movement between two transputers which perform the protocol processing of just the TP4 transport layer protocol. As with Braun’s approach, the transputers do not see the data itself. Diot claims that rates from 1300 to 3000 packets per second are achievable, from this approach, by using an instruction analysis, but the precise detail is unclear and the effects of process synchronisation do not appear to be included.

Diot and Roca have also been exploring XTP implementations using transputers. [Diot91] provides a useful analysis of XTP in contrast to the ISO TP4 transport layer processing, and shows some software-only performance figures for a range of shared memory schemes which, like that from the INMOS implementation, appear to expose bottlenecks in the shared resource. This is a little less serious, though, since their eventual objective is to migrate some of the work to hardware processing and application-specific integrated circuits. Their XTP implementation takes the C-language KRM implementation of XTP for the Unix operating system and adapts it to the
transputer environment, replacing all _mbuf_ references and re-implementing the timers and interrupts used to communicate between the separate XTP state machines. The user and network interface processes are the only two which manipulate the user data, exchanging it via a shared memory area which constrains the two processes to be placed on the same transputer. Apparently, the protocol processing is also performed on the same transputer, but this is not stated. Two such collections of processes, each running on one transputer, are joined together by a third transputer which provides buffering and connects the output of each TP4 process to the input of the others; this configuration is very similar to the TCP-to-TCP measurements reported in section 4.4.2 of this thesis. Throughput of 4.75 Mbits/sec (or nearly 600 kbytes/sec) are reported for large user segments of 2 kbytes and above - under half of that which my TCP implementation attains.

Two other research teams have reported the use of transputers with OSI protocols, although they have not provided any performance figures in their publications. S. Spahni [Spahni91] shows an architecture for processing the upper layers of the ISO stack, although he has not implemented the transport layer or below. Carchiolo [Carchiolo91] has also written a proposal for an architecture for processing the link layer of the ISO stack, which bears a very strong resemblance to Zitterbart's work. The code, even for this one layer, appeared incomplete when it was reported.

Most transputer development software supplied until recently has been compiled to run directly on transputer hardware, under support of a file server program which runs on the host computer into which the transputer is plugged. Similarly, binary transputer programs are either run on standalone hardware (as embedded systems) or in the manner described above, in which case they can access the host's facilities for screen, keyboard and file input and output. The most common access mechanism to such facilities uses a transputer card plugged directly into the expansion bus of a personal computer, workstation or minicomputer. There is usually a physical limit to the number of transputer cards which may be accommodated and individually controlled within each host.

I demonstrated in [Pee188] that part of the INMOS alien file server could be run on a personal computer containing an Ethernet card, and accessed from another host containing the screen, keyboard and file handling part of the server, across a local area network. At that time, personal computers were slow, and the TCP/IP software libraries provided for network access were single-tasking. These factors prevented more than one transputer from being hosted in each personal computer.

More recently, Luigi Rizzo has demonstrated the same facility running on Intel 80386-based personal computer technology, using multi-threaded TCP/IP software [Rizzo93]. He is therefore able to accommodate several transputer cards inside each personal computer - far more economical. Apart from this, and upgrading from the alien file server protocols to the newer Iserver ones, our networked servers are similar in behaviour. Rizzo's performance measurements include booting a transputer from a remote DecStation at 75 kbytes/sec, writing from the transputer to _/dev/null_ on
the DecStation at 45 kbytes/sec, and round-trip transaction times of 7 ms for empty transactions and
11 ms for 512-byte transactions. These figures will be compared with my all-transputer approach
in Chapter 5.

Brendan Murphy [Murphy94] has developed the networked file server concept further, building on
the networked-PC hosting mechanisms of Rizzo. Murphy's Enhanced Transputer Server
Architecture (ETSA) provides additional facilities such as user-installable server extensions,
multiple virtual Xterminals for each transputer program running across the network, a resource
location mechanism (to search for particular transputer configurations anywhere on the network),
and a choice of two implementation technologies - standard TCP/IP libraries and an ANS Aware
transaction-processing library solution. Performance for file reading was measured at
200 kbytes/sec, both when running the transputer directly plugged into an Intel 80286-based PC
and when accessing the same PC using the ETSA system and standard TCP/IP software from a fast
workstation such as a Sun4 or an Intel 80486 personal computer. Use of the ANS Aware transport
service roughly halved the performance. He also supplied some compilation timings, which
showed a local Intel 80386-based PC outperforming the networked version by between 10% and
80%, depending on the power of the remote server host.

Two other discussions of networked access to transputers have been published. Tom Hintz used
modems and commercial PC console remote control software to access PC-hosted transputer
networks in Australia [Hintz89]. This concept was flawed since modems were not available with
error detection and correction facilities in 1989. Isik Yigit [Yigit93] has produced a networked
server similar to those above, but just using the UDP datagram communication protocol [Postel80 -
RFC768], rather than TCP, and with no error detection or retransmission mechanism. This
approach is doomed to failure, even on lightly loaded local networks.

2.7. Summary of the Literature Survey

The previous sections highlight a number of high-performance implementation techniques for
layered protocols such as TCP/IP and the ISO stack. Almost all fall, however, into two categories:

The first category is all those protocol implementations which use a software shared-memory
model for passing all the user data between the application process and the network. These all
appear to present a bottleneck within the memory management code which controls access to the
shared buffers, typically achieving performances in the 300 kbytes/sec to 600 kbytes/sec range. In
most of these schemes, there are problems accessing the memory manager from more than one
transputer, which limits their parallelism. They also do not provide general communication
harnesses to permit access from many applications, although the lack of parallelism makes this
difficult, too.
The second category of protocol implementations is built around special-purpose multiple-transputer hardware which incorporates shared memory accessible from some, or all, of the processors. This rigid structure constrains the software configuration, and therefore the load balancing, and makes it difficult to attach other transputers running the application processes. Indeed, most of the architectures in this second category are designed as front-end processors for conventional workstations. Interestingly, the maximum transmission rate reported by most of these schemes is about 3000 packets per second, which is similar to that which I achieve and report in section 4.4.2. This figure is used almost universally to measure the throughput of these machines, often because the network interfaces are not yet implemented, or because they are being used to project ahead to the architectures' behaviour on faster networks, and tend to be quoted when transmitting very small packets. They are a little dangerous, however, because the shared memory synchronisation effects will not scale well towards higher data rates, and the cost of receiving more data into the shared memories will also increase. Indeed, all they do tend to measure is the performance of the packet transmission code - typically a loop of only a few dozen instructions!

Falling between these two categories is a gap which this thesis attempts to bridge. The gap is for an architecture which connects transputers running application code to an Ethernet, and which is capable of delivering almost the whole Ethernet bandwidth to one application, or of sharing it evenly between several. By constraining our requirements to a single 10 Mbit/second Ethernet, the network transmission rate is achievable using a single transputer link, and this transforms the project into one where using transputers throughout, with no hardware assistance, becomes attractive. Also, since each application will be interfaced using one of its serial links, it makes sense to continue this trend throughout the rest of the architecture. Performing the protocol processing in many separate processes facilitates load-balancing, provided that the processes can be joined together in such a way that performance does not suffer. Finally, cost and complexity are reduced by avoiding the need for the shared memory hardware. Interestingly, by increasing the processor power from that of first-generation transputers to that of the new T9000, and exploiting the new virtual communications channels, it is suggested in section 8.2 of this thesis that the performance should track that of shared-memory architectures which are built around T9000s and are suggested in several of the papers referenced.

I have found no references to most of these objectives in my literature survey. Although the figures are difficult to compare fairly, the TCP/IP communication performance of my software appears to match that of the best alternative approaches.

- 47 -
3. Research and Experimental Objectives

The overall objective of the research has been to design, implement, test and characterise a suite of highly-parallel TCP/IP protocol-processing processes, and to explore their applicability and usefulness in a number of demonstrator application programs.

To this end, it was decided to implement the TCP/IP protocol family across a number of pipelined processes, with each performing the processing associated with just one layer. The smaller number of layers associated with TCP/IP compared with the ISO stack is immaterial to the conclusions reached. The latter would have a higher latency as packets are passed across more process boundaries, but the techniques developed to speed the TCP/IP processing would be equally relevant in the former, where the parallelism obtained from processing a stream of packets through the pipeline of ISO processes would also keep all these processes busy in the most demanding circumstances. In addition, since TCP/IP implementations are much more common, developing a native transputer version would be of more immediate use to the research and industrial communities, and it would be easier to verify the parallel implementation against other manufacturer's more conventional serial ones.

This objective has been broken down into sub-goals:

(a) To provide high-performance network interface code which allows programs running on transputers to communicate with applications running on other networked processors. Its performance should ideally be capable of enhancement using parallel processing techniques, hence ...

(b) To investigate whether the communicating process model of parallelism is capable of implementing layered protocol processing with acceptable performance, modularity and flexibility of hardware architecture;

(c) To provide performance measurements and comparisons between protocol processing code running on different transputer-based configurations and on conventional processors running commercial networking code;

(d) To demonstrate a solely transputer-based hardware architecture which resides on a local area network and provides the standard range of INMOS software development tools to users logged in to hosts elsewhere on a local area network;

(e) To develop a low-cost transputer board capable of hosting several individually-resettable trees of transputers - necessary so that (d) can load, analyse and debug user code as commanded;
To demonstrate a networked instrumentation system which uses embedded transputers and the parallel TCP/IP networking software.

The following activities have never been part of the objectives of the project, since they militate against the compact embedded system approach:

- To build a front-end processor capable of off-loading the processing of network protocols from a more conventional workstation architecture; and
- To use vast amounts of hardware assistance to speed the throughput of packet data, such as the shared memory hardware inherent in [Braun91b] and [Chan94].
CHAPTER 4

4. A Parallel TCP/IP Software Architecture for Transputers

The three main forms of parallelism described in section 2.3.1 could be used in a distributed implementation of a layered protocol such as TCP/IP. Taking each in turn:

**Geometric Parallelism**

The data spaces for a number of disjoint TCP streams are independent, and therefore the processing for each could be performed in parallel in separate processes. Likewise, the transmit and receive activities of IP-level processing are independent. There is little scope for data-space separation within the processing at the TCP level since the TCP activities are specified as one large state machine, although section 8.1 of this thesis reports that some recent investigations into subdividing the TCP process into transmit and receive halves have been made.

Double or triple buffering of the communications in the various pipelined protocol processes (see below) are forms of geometric parallelism, too, if one considers the buffers in states of reception, transmission and intermediate processing as operating on separate data spaces.

**Algorithmic Parallelism**

A layered protocol lends itself to pipelined implementation since the whole concept involves successive manipulations of the streams of data flowing across a network. In the case of TCP/IP, the TCP, IP and low-level hardware layer processing could be performed in a pipelined fashion. Detractors will complain that the interprocess communication and memory-to-memory copying between pipeline stages reduce throughput and increase latency, e.g. [Clark82]. This is not true if the communication is run in parallel with the processing of successive packets (easily achieved with the transputer's direct memory access communication links) and if the throughput of the pipeline, dictated by the local area network transmission rate, is much lower than that of the transputer's memory bandwidth (which is certainly true for Ethernet). Latency remains an issue, of course.

As an enhancement to passing the complete data stream through the protocol-processing pipeline, it would also be possible to keep it separate and to run it through a parallel pipeline. This concept is well-illustrated in each of figures 13, 14 and 15, but each of them also shows that these shared memory areas must be accessed by each of the protocol-processing stages, and this virtual shared memory upsets the synchronisation and parallelism within the remaining processes. Another snag with this scheme is that several transputer links are used to connect to the data memories, and this
reduces the number available for the main pipeline and prevents them from forming the tree-structured distribution scheme which is developed below.

Process Farming

Rather than pumping network packets through a pipeline of processes, it is conceivable that a central controller could pass packets to processes which would perform one layer's worth of processing, and then accept them back for redistribution to another. This concept might have some attractions on the new T9000 transputer, which provides a fast virtual routing mechanism - see section 8.2. On the first-generation transputers used in this research, the communication costs of this farmed approach would be far higher than in a pipelined implementation. The need to remember the state of each active stream would require more communication, or would restrict the destinations for farmed packets to exactly the same processors which could better be connected in a pipeline. Only by using shared memory between processors - and therefore destroying the hardware simplicity which this project achieves - could performance in a process-farmed scheme be maintained above the pipelined version.

Alternatively, we could consider processing each incoming or outgoing packet completely in one of several workers. In this way, no data would need to be passed from processor to processor, and there would be no communication delays as the packet (or even just its headers) was passed around. Unfortunately, if we were not to constrain each packet in a TCP connection to be processed in the same process, all the state information for the stream would have to be sent to the process chosen to deal with each packet. Since this could only be done once the TCP stream had been identified, the extra latency would be unattractive.

The parallelisation mechanism eventually selected is a mixture of pipelining and geometric decomposition. Each TCP stream is processed in a pipeline of processes; outgoing packets propagate from the user application, through TCP, IP and the Ethernet controller processes, onto the network. Incoming packets retrace this path along a parallel pipeline. Should an application require more than one simultaneously-open TCP connection, it is simply connected to two or more TCP processes which, in turn, share the same IP stream. This static process structure directly mirrors the occam model and the design of the transputer itself.

Because transputer links are restricted to four per processor, it becomes necessary to introduce channel multiplexing processes between the TCP process and the IP process. These can fan-out the TCP streams by a factor of three at each extra level of transputers, also performing processor-intensive computations (such as the TCP checksum calculations and checking) using the additional power of the new processor.

Two such types of TCP routing process have been written. A single copy of a root router process resides next to the IP process, as shown in figure 16, and keeps track of where to send each
received packet. One root router is always required between the IP and TCP layers. Higher routing layers (the tree routers) simply provide the fan-out for larger configurations and carry their share of the checksumming load. It is even possible to perform the transmit and receive checksumming for each connection in different router processes on separate processors. This may improve load-balancing on very small configurations. Tree routers are optional, provided that the root router is configured to perform all of the checksum computations.

All processes, from the IP layer up to (but not yet including) the TCP layer, are themselves geometrically split into transmit and receive pipelines. These halves are largely independent, but the two halves of the IP and root router stages occasionally exchange information (usually when opening and closing connections) and this information has to be passed across in a deadlock-free fashion.

---

**Figure 16 - the full TCP process architecture**

Careful attention was given to the design of the channel interfaces between the TCP processes and the user's application processes. Almost all TCP implementations follow the scheme introduced in the Berkeley versions of Unix, which use "sockets", a specific (and rather clumsy) procedural interface. Unfortunately, on a parallel machine, synchronous sockets introduce an unacceptable latency since each socket call must return its result before the next call may be issued, and this precludes many instances of parallelism which could otherwise have allowed the application to be doing something else. Asynchronous sockets are also available in some conventional implementations, but are very rarely used, as evidenced by the bugs still remaining in many recent versions of the SunOS Unix socket library [Grabette94]. Error handling in this form of interface is very difficult to achieve without re-introducing synchronism, and thus asynchronous sockets were not deemed suitable for a parallel TCP interface.
Instead, a complex set of six channels is used to connect each TCP process to the user application, as described in [Peel92b, section 2.2]. This has the benefit that the TCP transmitter may be driven from the user process using a standard blocking channel to regulate the flow of messages, in parallel with the pair of channels which pass data from the network to the user process (and regulate its flow). At the same time, another channel returns asynchronous status and diagnostic information, and a sixth channel allows commands to be sent to the TCP process or to lower layers, to facilitate control of the whole system. This scheme has been shown to work well in practice, as well as being easy to write into the user application.

Ancillary to, and rather simpler than, the TCP (connection-oriented) protocol, is the UDP or datagram protocol. UDP packets may be processed by UDP processes which share the TCP routing tree and communicate with user applications along separate channels from the TCP ones. Applications may specify to the UDP process (and thence to the root router) which types of UDP packet interest them. The location of the UDP process is shown in figure 16. This facility has been used in a Teletext server, which captured pages as they were broadcast and distributed them around the Department using the local area network. Since all the data representing a page of Teletext could be fitted into a single UDP packet, and since retries of lost packets could easily be requested by the client process, UDP was much simpler to use in this application than TCP. See [Edwards93] for further details.

Notice that Sun Microsystems' Network File System (NFS) [Sun89], which usually relies on a datagram transport service, could be implemented in a process which contains UDP code and the extra file-service functionality, as well as a suitable user interface. Notice also that the conventional style of synchronous file-server interface used by the INMOS servers and (indirectly) by C-language file access routines would present a far higher latency than a well-designed channel-based interface to parallel occam processes.

At the bottom of the protocol hierarchy is the Ethernet hardware driver process. This is necessarily rather sequential (apart from some double buffering at its interface with the IP process) because it has to perform the packet management strategy required by the Ethernet controller device. Currently, this process is written for the AMD LANCE chip set, as implemented on two commercial boards manufactured by INMOS. A simpler driver process for a Fujitsu NICE controller has also been developed. This driver still retains some sequential features, but the Fujitsu device is far more amenable to parallel operation. It is irritating that only very minor changes to this device would allow it to perform its transmit and receive operations fully in parallel, but no manufacturer appears to have recognised these benefits yet.

The hardware driver usually just operates as a pipeline between the local area network and the IP process. When the driver is started, however, it has the important extra capability of being able to read a special code file which is stored on a Unix file server, and pass this code to the other transputers in the protocol-processing hierarchy, thus booting them from code stored on the
network. This makes it very easy to update the software of embedded systems in exactly the same way that most workstations are managed.

4.1. Detailed Code Structure

This section presents the structural detail of the major processes which make up the full TCP/IP architecture on transputers. The simplest collection of processes capable of supporting TCP is shown in figure 17. We shall examine the processes, starting at the left-hand side.

![Figure 17 - a minimal set of TCP/IP processes](image)

4.1.1. Low-Level Ethernet Control Software

Experience has been gained of three separate styles of Ethernet hardware during the implementation phase of this research. Two designs have been based around the AMD LANCE (Am7990) chip-set, one using a T222 16-bit transputer and one using a T414 32-bit transputer. The third is based on the Fujitsu NICE (MB 86960) device. All three are interchangeable with respect to the higher-level software processes (IP and above), although obviously the low-level drivers running on the Ethernet controllers require programming differently in each case. Particular features of the hardware designs are discussed in Chapter 6.

My first driver was written for a T414-hosted Lance controller which was built as a student project (see section 6.3). With 2 Mbytes of memory on this board, it was possible to run other processes in parallel with the Ethernet control process. The driver only required very minor modification, mainly to word-length addressing issues, before it could be run on a T212-based IMS B407 Ethernet transputer module (TRAM). In this case, the 64 kbytes of memory available to the 16-bit processor were only sufficient for the driver code plus a minimal buffer arrangement, and no other processes would fit on this TRAM. The same applies to the IMS B431 TRAM, which is physically far smaller than the B407, but is otherwise identical in capacity and behaviour.
A recent investigation, motivated by the desire for a cheaper Ethernet interface, identified the Fujitsu NICE controller as a superior product which is capable of simple integration with 16-bit transputers. It is discussed in section 6.4, and its controlling software in section 4.1.1.3.

Ethernet is a 10 Mbit/sec transmission medium, using Carrier-Sense Multiple Access (CSMA) techniques to share this bandwidth between all the connected nodes. If one node were to be involved with every message on a fully-loaded Ethernet, the network controller would have to process a mixture of received and transmitted packets, but either style might predominate in the extreme cases. The permissible packet sizes on Ethernet range between 64 and 1518 bytes, plus a packet preamble of 64 bits and an inter-block gap of 9.6 μs or 96 bits [Digital80]. This means that approximately 14 880 shortest-length packets may be transmitted per second, carrying 952 kbytes per second of packet data, allowing for inter-block gaps and error detection bits, or 812 longest-length packets, carrying 1.23 Mbytes per second of packet data. Due to contention between the various nodes attempting to transmit on the Ethernet, and the speed of the processes actually generating or storing the transmitted data, a more realistic maximum data rate on a loaded Ethernet is nearer a third of its maximum capacity.

The activities performed by the low-level hardware driver processes include:

- On power-up, the Ethernet controller and all attached transputers (which will subsequently run higher-level code such as IP and TCP, as well as the main code for this application) are reset, readying them to receive their code along a link;

- The Ethernet controller is booted (from the motherboard EPROM or from its own ROM, depending on its configuration), initialises itself, and sets up its data buffers and free-buffer lists. The 48-bit Ethernet hardware address of this node must be supplied as a parameter to the driver, and therefore identifies the device uniquely on the Internet.

- The driver runs all its transmit and receive processes. If the IP address of the node has not been specified as a parameter to the driver, or if network booting of all other transputers connected to the Ethernet controller is required, it directs all received packets to its Boot process.

- If the Boot process is being used, it performs an RARP interrogation of any boot servers on the network, thereby discovering the node’s IP address. It then opens a file on this boot server, using a filename corresponding to its IP address, and reads the contents using the Trivial File Transfer Protocol (TFTP). These are sent directly to the link which connects all the other transputers to the driver, thereby booting them. Upon completion, the newly-loaded transputers start running their protocol-processing code, and the Ethernet controller directs all received packets to the IP process. The boot process is illustrated in figure 18.
On hardware which supports programmed resets of the subsidiary transputers (such as our Fujitsu NICE board), it is possible to switch back to the boot mode, to reset the subsidiary transputers and to re-load them. In this case, it is also possible to reset the subsidiary transputers in analyse mode, and to use the TFTP mechanism to extract a dump of the contents of all the transputers, aiding remote debugging of the networking software.

Both the DTR2 motherboard (see section 6.2) which supports the INMOS Ethernet TRAMs and our Fujitsu NICE board (see section 6.4) have eight light-emitting diodes (LEDs) which may be turned on or off by the T222 transputer which controls the booting process. It has been found nearly essential for the various stages of the booting process to be reflected in these LEDs, and for the cause of any failure to boot to be indicated by various static and cyclic sequences of light outputs. This is because it is very difficult to extract booting information from such an embedded system in any other way, without connecting up a development system. Most problems at this stage are caused by errors in the database of host and Ethernet addresses, and by read permission denials on the boot code file.

4.1.1.1. AMD LANCE Control Software

The LANCE controller provides a Direct Memory Access (DMA) interface with the attached transputer for both incoming and outgoing Ethernet packets, storing or collecting them via special rings of buffers which have been set up in shared memory and are accessible by both devices. See [AMD86] for full details. Control communication between the LANCE and the transputer is by memory-mapped reading and writing of the LANCE registers, always initiated by the transputer. For signalling initiated by the LANCE, an interrupt is generated which the transputer can use to start interrogation of the LANCE registers.
The maximum number of LANCE interrupts seen each second by the node processor could exceed 14,880, since one is needed for each packet either transmitted or received, and more can occur with some buffering strategies or during error situations. The interrupt process could see twice this level of activity, however, since it sends a response to each interrupt from the rest of the software back to the LANCE as well. More usually, long Ethernet packets will be sent when traffic levels rise, and the number of such packets will drop to only about 1000.

The LANCE has been very carefully designed to minimise the programming effort required from implementors on conventional architectures. Its user interface also strongly reflects its complicated internal architecture. As a consequence, it has proved impossible to produce an occam interface to the LANCE which deviates substantially from the sequential flow charts which are provided by AMD as a basis for sequential designs. However, much of the algorithm which AMD maps out has been separated into parallel input and output buffer managers and a LANCE buffer manager, for tidiness and readability rather than performance, since the hardware prevents them from performing much activity independently. These processes are pictured in figure 19. There is also little scope for parallelism in the interrupt process.

![Figure 19 - low-level occam interface to the LANCE controller](image)

The LANCE process is structured around the buffer management strategy which is dictated by the hardware. Each network packet is only copied twice - once by the transputer communication along channels `rx.out` or `tx.in`, and once by the LANCE when it is transmitted to, or received from, the Ethernet.

When a packet is received along channel `tx.in`, it is stored directly into a LANCE transmit memory buffer, with the input channel blocking until a buffer becomes ready. The choice of which buffer to use is made by requesting a free buffer descriptor from the "tx fifo" controller. After
storage, a message is then passed back to the “tx fifo” process, telling it that the buffer is ready for transmission. The buffer is thus turned over to the LANCE, which transmits it in due course and interrupts the transputer when it has finished. An interrupt wakes up the “interrupt” process which has been executing an occam input primitive on the special Event channel. This process determines the required action; after a successful transmission, it sends a message to the “tx fifo” process to let it know that the buffer is now free again. Errors are detected and dealt with separately. After data has been received from the Ethernet by the LANCE, it is stored in one of the receive buffers and an event is used to signal its arrival to the “interrupt” process. In turn, this sends a message to the “rx fifo” process, which buffers the request and in due course passes it to the “receive” process, which blocks until the \texttt{rx.out} channel accepts the packet - again, using a single occam output primitive to copy the buffer contents directly to the channel.

The interrupt process just described contains a fifo buffer of its own, to ensure that interrupts can never be lost by the “interrupt” process blocking on its output channels (see [Burns88, section 3.6.2] for descriptions of sacrificial buffering, the three-way handshake and fifo processes).

Care has been taken to ensure that all packets of Ethernet data managed by this group of LANCE control processes are only ever communicated or copied once, which minimises data bus utilisation and keeps throughput high. The use of multiple “receive” or “transmit” processes could enhance throughput by sharing the communication load across multiple transputer links; the “fifo” process would have to be enhanced to record ownership of each buffer and the order in which buffer descriptors were passed to and returned from the LANCE rings. In practice, a 20 Mbits/sec transputer link has always proved adequate to carry the full Ethernet traffic, and such sophistication has not been necessary. Note, however, the effect of Direct Memory Access between the LANCE and the transputer, which slows transmissions on the link by delaying its access to memory. See section 4.1.1.2 for more details.

One interesting feature of the transputer which can usefully be employed at this stage is that the receiver and transmitter processes (“receive” and “transmit” in figure 19) interface to the IP receive and transmit processes at the next higher level via just two occam channels, \texttt{tx.in} and \texttt{rx.out}. It is therefore more efficient, in terms of memory bandwidth utilisation, to separate the LANCE controller from the TCP/IP processing at this point using DMA-driven links rather than along on-chip communication channels, which use direct memory copying under processor control. This is completely at variance with conventional design principles - where communication is expensive - and is easily demonstrated by writing a packet generation process (which simply transmits a pre-defined array along a channel) and then placing it on the LANCE transputer in parallel with “transmit” and connected to it via a soft channel, and alternatively on a separate processor, connected to “transmit” via a link. The link-connected version will out-perform the former method in all cases where CPU (and therefore memory) utilisation is high at the receiving end.
Thus a 16-bit transputer with a LANCE chip and minimal buffering provides a more efficient Ethernet interface to a 32-bit transputer (which runs the rest of the protocol-processing code) than the 32-bit transputer could obtain by being directly connected to the LANCE and doing the job itself.

4.1.1.2. The Effect of Direct Memory Access on Link Throughput

Many peripheral interfaces employ Direct Memory Access (DMA) to request the use of the processor’s data bus when they have just received data or when they are just about to transmit. The Am7990 LANCE Ethernet controllers on the INMOS Ethernet TRAMs (the IMS B407 and the IMS B431) access the data buffers which they share with their host transputers in this way. One snag with this strategy is that transputer links cannot run at their maximum specified speed if they are accessing memory which is subject to DMA interruptions.

Consider the case of the aforementioned TRAMs, which are logically identical, and are based around a T222 16-bit transputer, an AMD LANCE Ethernet controller and 64 kbytes of static memory which is shared between the two. Network activity at the full Ethernet rate of 10 Mbit/sec causes the LANCE to grab DMA control of the T222’s data bus about every 12 μs, and to hold on to it in most cases for about 4.8 μs - about 40% of the time [AMD86]. When using the B407 to pass data from a link onto Ethernet, this has the effect that there are very frequent occasions when the link on the T222 processor becomes stalled waiting to perform a memory access, even when a multi-byte channel transfer is in progress. Hence the T222 link, which is capable of transferring more than 1.4 Mbytes / second when the LANCE is not running, slows down to only about 0.8 Mbytes / second when network activity is at its highest - making its link the bottleneck, rather than the protocol processing further up the transputer pipeline or the network itself. I presented an early report of the effect in [Peel92a]; the discussion here updates this to 20 MHz devices and to the newer T225 processor.

Since link activity on the Ethernet TRAM also implies processor activity (to manage events and to keep track of the buffer rings), I performed a simplified experiment to demonstrate the effect of DMA on the speed of the links of a T222 processor, independently of the other factors. Using a suitably-configured programmable logic device (PLD), I asserted MemReq, waited for the T222 to respond with MemGranted, waited for an additional 4.8 μs (to simulate the LANCE’s DMA activity), deasserted MemReq, waited for 7.2 μs (to give a 40% MemReq duty cycle) and started again. The timing was done by counting cycles of the processor’s ProcClockOut (96 and 144 cycles, respectively, in the case of a 20 MHz transputer). In these experiments, the transputer’s off-chip memory, containing the buffers which the links transmit and receive, was 1 wait-state static RAM accessed through a 16-bit data bus; T222-20 and T225-20 processors were employed, and the link being measured was connected to a T8000-20 which only used its fast internal memory. The experiments were repeated at 5 Mbps, 10 Mbps and 20 Mbps link speeds and with the 16-bit
transputer's data buffer alternatively stored in its on-chip memory as well as its slower off-chip memory. The data was encapsulated in an occam `BYTE; INT16 : [ ] BYTE` protocol with 1400 bytes in each packet, to simulate a typical Ethernet load and to reflect the protocols on the channels `tx.in` and `rx.out`; low priority was used throughout; the compiler was TDS3 and all compiler checks (for range checking, etc.) were enabled.

The rates of data exchange when DMA was in use and when it was not in use are given in table 1 (for the T222 processor) and in table 2 (for the T225 processor).

<table>
<thead>
<tr>
<th>Link speed</th>
<th>Buffer location</th>
<th>No DMA</th>
<th>40% DMA asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T222 to T800 (kbytes/sec)</td>
<td>T800 to T222 (kbytes/sec)</td>
<td>T222 to T800 (kbytes/sec)</td>
</tr>
<tr>
<td>5 Mbps</td>
<td>on-chip</td>
<td>433</td>
<td>452</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>415</td>
<td>452</td>
</tr>
<tr>
<td>10 Mbps</td>
<td>on-chip</td>
<td>795</td>
<td>902</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>764</td>
<td>902</td>
</tr>
<tr>
<td>20 Mbps</td>
<td>on-chip</td>
<td>1361</td>
<td>1663</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>1274</td>
<td>1663</td>
</tr>
</tbody>
</table>

Table I - the impact of DMA on a T222 transputer's link throughput

Notice that the transfer rates only depend marginally on whether DMA is active when the data buffer is in on-chip memory - since DMA does not affect this memory. It is unclear to me why the T222 transfer rates at 20 Mbps should increase slightly in the presence of DMA activity - the order of on-chip memory accesses appears to change in this case.

The T222 link transmit engine is unbuffered and pauses for memory access between every pair of bytes transmitted. It is thus always slower than the receiver, regardless of the link data rate or DMA activity. A high DMA load on the T222 bus affects its link transmission rate in almost the same proportion as the duty cycle of the DMA assertion - 40% in our case. The effect reduces at the lower link rates since memory is accessed less often then.

The T222 receiver has double buffering and therefore runs faster than the transmitter, tolerating slower memory and DMA interruptions rather better. Indeed, it runs as fast using external memory as with on-chip memory when DMA is not in use. Only when using DMA and off-chip buffer areas does a degradation of up to 30% occur - less for the slower link speeds.

The T225’s link transmit engine is double buffered, like its receive engine, and both engines transfer data at the same rates, except at 20 Mbps when the receiver is fractionally faster. With DMA, and when using on-chip memory, there is no change in the data rate at the lower link speeds, and only a marginal reduction at 20 Mbps. The T225 link transmitter is faster than the T222’s
when they are running at 20 Mbps and DMA activity is present, up from 829 kbytes/sec to 1155 kbytes/sec, thanks to the double buffering. The T222 and T225 receive rates are identical, at 1157 kbytes/sec, in these circumstances.

<table>
<thead>
<tr>
<th>Link speed</th>
<th>Buffer location</th>
<th>No DMA</th>
<th>40% DMA asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T222 to T800 (kbytes/sec)</td>
<td>T800 to T222 (kbytes/sec)</td>
<td>T222 to T800 (kbytes/sec)</td>
</tr>
<tr>
<td>5 Mbps</td>
<td>on-chip</td>
<td>452</td>
<td>452</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>452</td>
<td>452</td>
</tr>
<tr>
<td>10 Mbps</td>
<td>on-chip</td>
<td>902</td>
<td>902</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>902</td>
<td>902</td>
</tr>
<tr>
<td>20 Mbps</td>
<td>on-chip</td>
<td>1712</td>
<td>1748</td>
</tr>
<tr>
<td></td>
<td>off-chip</td>
<td>1712</td>
<td>1748</td>
</tr>
</tbody>
</table>

*Table 2 - the impact of DMA on a T225 transputer's link throughput*

The conclusion of this experiment is that the link throughput is dependent on the amount of DMA activity on the processor - especially with the un-buffered transmitter of the T222. In order to sustain the full data rate of Ethernet along one link (1230 kbytes/sec after stripping the packet framing), the T222 would need to be run with no DMA at all. Since the IMS B407's LANCE Ethernet controller uses external memory with DMA, the measured link throughput falls to a simulated 829 kbytes/sec for packet reception via a T222 processor - similar to that seen in a real networked application [See section 4.4.6]. To overcome this, two or more links could be used concurrently, alternating network packets between them using a special triple buffer which would attempt to keep both links in use whenever possible. Because it is not desirable to allow packets to overtake each other, synchronisation between the links would limit the benefits of this parallelism, especially if a mixture of long and short packets were being carried. The parallel-link scheme also reduces the number of links available to pass data further up the protocol-processing pipeline, due to the 4-link valency of the transputer. Even to attain the 829 kbytes/sec figure, care must be taken that the T222 may always initiate a transmission whenever it is ready - so the T800 in this example must employ a lightweight multiple buffering strategy and greedily accept network packets.

Simpler than this, the double transmit buffering in the newer IMS T225 transputer raises the thresholds discussed above, so that the lowest sustainable rate rises to 1155 kbytes/sec. This approaches the maximum rate of Ethernet itself, and is symmetrical for each direction of transmission, but is still short of the 1230 kbits/sec required. Neither double buffering nor the increased speed (of a 25 MHz or 30 MHz transputer) avoids a substantial degradation of the link throughput when DMA activity is high, but the faster transputers would achieve Ethernet data rates,
if they were compatible with the B431 or B407 TRAM hardware. A technique which would be more effective would be to separate the LANCE controller from the T222 data bus with dual-port memory, thereby eliminating the need for DMA entirely. Alternatively, we need to find a more suitable Ethernet controller chip, which is reported in the next section.

4.1.1.3. Fujitsu NICE Control Software

The Fujitsu NICE Ethernet controller (MB 86960 - [Fujitsu93]) provides an I/O-mapped interface to a private 64 kbyte packet buffer memory, thereby avoiding the need to store packets in the transputer's memory. This simplifies the access arrangements from the controller, and avoids the need for rings of FIFO buffer descriptors as seen in the LANCE. When attached to a 16-bit transputer, it also reduces pressure on the processor's addressing capability, to the extent that a very minimal TCP protocol stack (with small TCP buffers and no pipeline buffers) may just be accommodated in the 64 kbytes which the transputer can address. See figure 78 for the block diagram of an Ethernet controller based around the T225 transputer and a NICE chip.

The NICE registers are designed for I/O-mapped access in an IBM-PC architecture, and are therefore bank-addressed so that only eight I/O port addresses are used. Each of these addresses provides access to a 16-bit register. Unfortunately, many of these registers contain status information from, or provide control functions for, both the transmit and receive parts of the device. It is therefore impossible to write control information for the receive part from a receive process and for the transmit part from a transmit process, since the same memory address would be accessed in parallel by the two processes. Similarly, it is impossible to read status information for the receive part from a receive process and for the transmit part from a transmit process, since some of the status bits are reset when they are read, and this would therefore provoke the equivalent of a parallel usage error. Of course, the occam compiler would not be able to reason about this hardware side-effect.

![Figure 20 - the NICE controller process](image_url)
It is therefore necessary for a separate interrupt process to interleave all communications to these registers, as shown in figure 20. The transmit-data and receive-data registers are both 16-bits wide, and may be address-decoded to separate transputer addresses, thereby allowing them to be used in parallel without problems.

The combination of transmit and receive status indications into common status registers could be considered to be a design flaw of this device, making the "interrupt" process more dominant than if the transmit and receive processes were able to access their status registers independently.

Packets may be written one 16-bit word at a time to the NICE transmit register, prefixed by two-byte headers which specify their length. By using the clever address decoding scheme described in section 6.4, it is possible to make this port the destination for a transputer block-move instruction, as well as for a standard communication. Ethernet transmissions may therefore be performed using a single occam input primitive, as in figure 21:

```
[1024] INT16 NICE.tx.port : -- memory-mapped to one location
CHAN OF INT16 :: [] INT16 tx.in :
:
:
tx.in ? CASE
  xmit; length :: NICE.tx.port
  ... now tell NICE to send this
  led; INT16
  ... send value to diagnostic lights
  closedown
  ... terminate, ready for dump or re-boot
```

*Figure 21 - code transfers transmit packets directly into the NICE hardware buffer*

It is necessary to modify the protocol semantics of the tx.in channel from those of the LANCE controller, in section 4.1.1.1 above, to encode the length (in 16-bit words) of the packet itself in the first word transmitted along the channel. Apart from this, the NICE controller performs the whole transmission unassisted, with no need for any buffer or FIFO management code. The tagged protocol handles ancillary functions such as writing to the diagnostic LEDs (essential for feedback when a network boot fails) and for closedown operations.

Similarly, the NICE chip performs all receive actions, up to the delivery of received packets as a stream of 16-bit words from the receiver I/O port. Figure 22 shows that we have to determine the length of each packet, and then transmit it to the rest of the networking software using the rx.out protocol.
All that remains is to initiate transmit and receive operations. When the NICE controller generates an interrupt saying that a packet has arrived or a transmit buffer is available, the interrupt process passes a simple message to the transmit and receive processes. The tx.in channel also may carry tagged command messages, and the transmit process filters these messages and passes them to the interrupt process, since only the latter is responsible for accessing the NICE status registers. The status registers, transmit data register and the receive data register are logically separate, so may be accessed in parallel by the three processes described.

An overwriting buffer is required to prevent a deadlock cycle between the transmit and interrupt processes. It simply accumulates the total number of transmit buffers which are currently empty, and passes this value onward when the transmit process is willing to accept the information.

A programmable subsystem port on this card allows the NICE controller process to reset, reload, analyse and dump the memory of the subsidiary transputers, making this hardware far more flexible than the INMOS B407 and B431 products. In particular, it allows embedded system software to be developed using the networked facilities described in Chapter 5, and then loaded repeatedly into hardware which may be inaccessible to the programmer. This location-independence is highly convenient; only the useful diagnostic lights on the various transputer cards are not currently accessible to a system manager accessing the card remotely. Suitable software could easily be written to pass this status information across the network, too, using the UDP protocols whose code is already present in the network-booting software.

4.1.2. Multiple Buffering Strategy

When originally written, my pipelined IP and TCP routines were not capable of performing input, output and computation in parallel. [Peel89] showed some of my early attempts at to introduce this parallelism by incorporating shared buffers, and these techniques have now been retro-fitted to the original code in all possible locations. A selection of techniques and their effects are discussed in [Jones89b], and are summarised in section 2.3.3 of this thesis.
At first sight, the addition of extra processes to the pipelines may appear to impose a substantial communication overhead on the protocol management, and this is indeed apparent when the processes are all placed and executed on a single transputer. However, the tree-structured pipeline processes have been designed so that they normally run their processing and communication in parallel, with all the network data encapsulated in multiple buffers, and therefore most of the overheads of communication are avoided when the pipeline elements are spread across separate transputers and use the DMA capabilities of the transputer links. This is especially true when the input and output buffers of each stage of the pipeline are run at high priority - a standard recommendation [INMOS89b, chapter 14] which typically makes a 10-20% improvement to the pipeline throughput (see section 4.4 for an example).

Of course, the buffering policy adopted here has a latency which is higher than if the extra processes were not included - or if a shared memory architecture were chosen - but this is offset by the distribution of the processor-intensive activities across transputers and the improved load balancing. Most of the latency effects are hidden by the end-to-end synchronisation of individual TCP streams across the physical network, and by the sliding window buffering techniques used in the TCP processes.

One multiple-buffering scheme which I have not seen described elsewhere is that of a double-buffered prompter. It is common for a prompting process to be used as a sacrificial element on the output of a major process, such as a FIFO or my TCP process. The prompter (see figure 23) requests the main process for messages to pass on, has its request accepted only when that process has a message available, accepts the message by communication and then passes it on. The inclusion of a prompter prevents the main process from attempting to output and being blocked if the consumer is not ready, allowing it to service incoming tasks and requests from a number of sources.

![Figure 23 - the prompter mechanism](image)

Unfortunately, the intermediate communication referred to above adds to the latency of the channel if it is performed sequentially, as described. The delay can be avoided by double buffering, thereby overlapping the two communications completely, if need be. The occam code to do this is shown in figure 24:
4.1.3. The IP Process

The IP process performs the actions of a number of low-level protocols, including Internet Protocol (IP) itself, Internet Control Message Protocol (ICMP), and the Address Resolution Protocol (ARP). All these are primarily concerned with address translation between the unique 32-bit domain-based Internet address by which TCP knows all networked hosts, and the 48-bit flat address space of their Ethernet interfaces. When the IP process code starts running, it waits until it is supplied with the node's Internet address and its hard-wired Ethernet address by the low-level controller code. It also receives information regarding the sizes of the underlying Ethernet controller's transmit and receive buffers, and the maximum size of packets that the underlying network will accept (the Maximum Transmission Unit, or MTU).

When a connection is initially made from this station, the Ethernet address of the destination station is discovered using ARP, which also passes this station's Ethernet address to stations requiring it prior to initiating a connection request. Code to process received ICMP packets runs continuously, providing information about this station to others, and receiving status information about packets sent earlier.

The main work of the IP process is to add an Ethernet address onto outgoing packets, also providing a checksum of the header information as it passes. The former involves maintaining a table of translations, updating them if they change, and searching the table for every packet transmitted. Messages destined for a network other than the local one have the address of a suitable gateway added, which is found from a separate routing table containing references to known gateways. If the packet which TCP presents for transmission is too big for Ethernet or any
network section on the transmission path, the message is fragmented into smaller segments before it enters that section.

![Diagram](image)

**Figure 25 - the IP process buffering**

On the receive side, incoming Ethernet frames are stripped of their Ethernet address, routing information for subsequent responses is stored, and the IP header checksum is calculated and checked.

Since the transmit and receive activities are virtually independent, the IP code has been written as two separate parallel processes. They communicate occasionally to exchange routing and IP address table information, and the potential for deadlock has been avoided by careful buffering of all messages on these channels.

The transmit and receive processes are triple-buffered so that input, processing and output may be performed in parallel - see figure 25. Only providing two buffers in order to enforce the order of communication and thereby to provide a form of prioritisation, as discussed in [Peel89], was abandoned in favour of the simplicity of everything happening at once. When the transmit IP process is sent a packet generated by the receive IP process (typically in response to an ARP or ICMP request), a separate buffer which is not part of the triple-buffering scheme is used. This ensures that the main transmit pipeline buffers are not disrupted.
Currently, the ARP and routing table look-up algorithms in the IP transmit process are implemented as simple sequential searches. There is no need for more sophistication at present, because the embedded applications and network benchmarks run to date have required very few entries in the tables, and the set-up overhead of most other table searching schemes would dominate the timings. Should the IP code be used to implement an IP gateway, or a firewall between two networks, more attention would have to be applied to maintaining all the currently-accessed addresses in a better manner. Other issues relating to gateway implementations may be found in section 4.1.9, which deals with conformance of my software to the published RFC standards.

4.1.4. The TCP Router Processes

The purpose of the routing layer is to provide access upwards from the IP layer to all the transport-layer processes (mainly TCP and UDP), and vice-versa. It was necessary to design a scheme which:

- minimises diameter and latency - the routing structure must be crossed by all network traffic;
- supports transputers which have just four links - but yet must provide network access to a potentially large network of transputers;
- may be controlled in a distributed fashion - since each TCP or UDP process will be operating independently;
- provides a service which allows eager transmission (nodes may use as much bandwidth as is available, rather than a fixed share), but yet also provides fairness (nodes cannot exclude other nodes wishing to transmit, so all stations can obtain an equal share of the bandwidth if they want it);
- is deadlock free.

In addition, it was considered important to be able to spread the computation (of the routing algorithm and of the TCP checksum calculations) across as many of the routing processors as possible. Originally, the generation of TCP retransmissions in the routing processes was considered, although later this function was removed back to the TCP processes to reduce the actions necessary when an acknowledgement was received.

Many schemes for TCP packet routing were examined. Remember that all TCP packets will converge at a single point (currently the IP process), regardless of how many transputers are connected; some of the usual experiences of designing transputer communication networks which carry evenly-distributed traffic do not apply here.
A ring of transputers could be constructed, with one of them connected to the transputer running the IP process (as in figure 26, or even with one of them running the IP code in addition to passing messages around the ring). Considerations of performance, flow control and deadlock management of Welch's TRANSNET ring architecture are discussed in [Welch89a]. A ring of transputers is extensible in small increments, although since two links are used to form the ring itself, only two external links are added by each extra transputer. The diameter, and therefore latency, of the ring also increases linearly as the ring is expanded. The nature of the low-level routing algorithm on a ring implies that each ring node relays an average of half of all the underlying Ethernet traffic plus half of the flow-control packets passing in the opposite direction. This activity would have to be performed in addition to any checksumming or high-level TCP routing operations. Sadly, it is unlikely that the ring nodes in this application could be constructed from 16-bit transputers using just on-chip memory, due to the buffering required and the memory demands of the TCP router code.
Alternatively, the IP process could be connected to the TCP processes using a mesh, cuboid or hypercube arrangement, as shown in figure 27 and described in [Shumway89]. This is really just a two- or three-dimensional enhancement of the ring scheme, and many of the same observations apply. One significant difference, however, is that only transputers on the periphery of the mesh have free links for connecting to processors running TCP processes and user applications. This makes such multiply-connected schemes less efficient in terms of the number of processors used. Another difference is that the diameter, and thus latency, is considerably reduced. Since routing algorithms for meshes are usually non-cyclic (to avoid deadlock), it would be possible to eliminate some of the flow-control strictures of the ring and to run a mesh in blocking mode - with new messages flowing in as others were extracted. Attention would have to be paid to fairness if this behaviour were chosen.

![Figure 28 - TCP packet distribution using a tree structure](image)

Many of the benefits of the mesh are also obtainable from a tree structure, as pictured in figure 28. A tree is the most efficient way of fanning out connections from a single point (the IP process) to many leaf nodes (TCP processes), both in terms of the number of 4-valent transputers required and regarding latency. It is very suitable for running in a blocking mode, although it requires fair alternation at each stage of multiplexing in order that it provides fairness, as indicated for the mesh above. A tree structure may be considered as two parallel trees - one of multiplexers carrying packets towards the IP process, and one of demultiplexers carrying packets away from the IP process. The same sub-division could also be envisaged for all the other schemes, although some clarity (and therefore scope for process re-use) would be lost in most of them.
A solution using transputers connected via a crossbar switch (the IMS C004) was briefly considered - see figure 29. The INMOS C004 is a 32-link crossbar switch, with a 33rd link used to control the connections of the other 32. Only one link may be connected to any other, and only bidirectional paths may be programmed, so that the links' per-byte acknowledgement scheme continues to work as if the switch was not present. One snag with using a C004 in a many-to-one (and one-to-many) mode is that each connection to the IP process must be set up, the communication made, and the connection broken again, ready for the next packet to be sent. This requires a second path - maybe a tree or ring of links - in order to pass the re-configuration requests to the processor which controls the switch matrix! Thus, although the diameter would be very small (just one direct hop), the latency would be quite high. In addition, the C004 scheme does not introduce any extra processors on which checksum calculations may be performed, which would not allow as much parallelism in the processing of each TCP stream. The C004 was rejected easily once this was realised.

When the project started, the T9000 transputer had not yet been announced, and so it was not considered. Section 8.2 of this thesis looks, retrospectively, at what the T9000 and C104 packet routing scheme has to offer - low latency and greater flexibility in placing all the protocol processing processes across the architecture.

The final choice of TCP routing structure was the tree, and is shown in figure 30. In order to connect many transputers, each possibly carrying a number of TCP and UDP processes, a multi-level scheme was devised. This uses two types of TCP routers - a root router which resides next to the IP process, maintains the routing table and makes most of the routing decisions, and a tree router which performs simpler fan-in and fan-out tasks and sits at higher levels in the hierarchy. All of these routers may handle fan-ins of three (suitable for 4-valent transputers) or more (for fanning-out to TCP processes on the same processor).
It would be possible, of course, to remove the tree-structured routing scheme and to replace it with a ring-style (TRANSNET) one, a mesh-based or hypercube-based one (based on the Shumway description), a single root router plus C004 hardware facility, or even a T9000 and C104 solution, subject to the comments made about each of these above.

The TCP router processes each run a demultiplexer and a multiplexer in parallel. They are used to route incoming IP packets to the TCP process to which they are addressed, and vice-versa. The routers nearest to the network typically fan out the TCP streams along three channels corresponding to the fan-out obtainable using transputers with four links. The router(s) running on the same processor(s) as the majority of the TCP processes may fork to more than three channels since these channels are soft ones. One router between each TCP process and the IP process must perform the computationally-intensive checksumming operation on each packet passed in each direction. The choice of which router process performs this task is configurable to improve the load-balancing of multi-processor pipelines.

Each router is made up from an n-way multiplexer and a n-way demultiplexer running in parallel.

### 4.1.4.1. A Multiply-buffered Multiplexer

The n-way multiplexer with shared buffers uses three (or more) buffers per input channel. Each input channel is serviced by a separate parallel process which receives a packet, and then sends its buffer number to the central process which accepts it into a large ALT construct, thereby combining the input channel data streams (see figure 31). After processing the route information for this packet and possibly checksumming the data itself, the buffer number and stream number are passed to the parallel output sub-process, which sends the data onwards and sends the freed buffer number...
back along one of an array of channels to the relevant input sub-process. These return channels are decoupled by small buffer processes to eliminate the possibility of deadlock and to ensure that the output process is not held up unnecessarily. As the number of channels increases, the cost of the central ALT statement in this multiplexer can become a significant load within the whole TCP/IP pipeline, which may limit performance for any particular process configuration. In any case, it is necessary to implement a FAIR ALT [Jones89a] here to prevent one continuously busy channel from always being selected to the exclusion of the others.

![Figure 31 - a multiply-buffered multiplexer](image)

**4.1.4.2. A Multiply-buffered Demultiplexer**

Implementing an \( n \)-way demultiplexer with shared buffers is more complex. By providing three or more buffers, one may be used by the input sub-process, one by the main process (which determines the routing using a table look-up, and possibly performs checksumming) and the remainder by one or more of the \( n \) parallel processes which pass the packets to their selected TCP processes or to further routers in the tree. The input process is itself parallel, and implements double buffering, so that one buffer may be being filled whilst the number of the previous one is being passed to the main process. The parallel output processes incorporate FIFO queues to maintain channel activity at their outputs whilst helping to prevent their inputs from blocking the other streams. Passing the freed buffer numbers from the output processes back to the input one involves the use of another FAIR ALT construct, because even when just passing buffer numbers between processes, starvation of some streams would be detrimental. This FAIR ALT is expensive to execute, like the one in the multiplexer process, as the number of processes increases.

The TCP processes at the leaf nodes of the tree must be defined never to block for an excessive interval; otherwise, the whole tree of incoming router processes may itself eventually become blocked to the detriment of other active connections. This behaviour is easily achieved due to the windowing nature of the TCP protocol itself.
4.1.4.3. Routing Decisions in the Root and Tree Routers

The overall task of the tree of TCP routing processes is to multiplex packets passing from the TCP processes to the IP process, and to identify packets emerging from the IP process and to pass them to the correct TCP process. The routing decisions must be made in a distributed fashion, so that the routing tree may be placed across several transputers. In addition, all packets passing in each direction must be checksummed, and the checksum value either inserted in the outgoing packets or checked with the value found in the incoming ones.

One element of the protocols which carry TCP packets through the routing hierarchy is an INT32 quantity which contains the results of all routing decisions made above the point under observation. As a packet is passed through each multiplexing stage towards the IP process, the incoming value of this routing variable is shifted left by enough bits to allow a number (from 0 to the maximum number of channels multiplexed in this process) and then the channel number is added.

Figure 32 - a multiply-buffered demultiplexer

Figure 33 - the distributed TCP routing mechanism
Provided that the TCP processes generate packets with a routing value of 0, the value after passing into the root router will provide all the information required to identify the return path. This route identification scheme does not require all the TCP processes to be connected to the IP process via the same number of levels of routing processes, so long as no more than 32 bits of routing information is required along any path.

For example, in figure 33 above, the root router will find a route number of 73 in decimal or 100.10.01 in binary in each packet which it receives from process [TCP4] in the following way:

- Process [TCP4] sends a packet with a route value of 0 to the right-most tree router process.
- This tree router requires three bits to address its input channels, so it shifts the incoming route number (0) three bits to the left, adds in the number of the channel on which it received the packet (4 in this case, giving 4), and passes the packet on to the middle router process.
- The second tree router process needs two bits to address its inputs, so it shifts the incoming route value by this amount and adds 2 (making 18), before passing the packet on.
- The root router also needs two bits to address its inputs, so it shifts the incoming route value by this amount and adds the number of the channel from which it received the packet (1). The final value of the route variable is 73, which the process stores together with the local and foreign IP addresses and the local and foreign TCP port numbers for this connection.

Packets passing in the opposite direction are looked up in the root router's table, using the four fields listed above, and are passed out to the channel which is numbered by the least significant two bits held in the routing table. In the process of transmission, these two address bits are stripped off, leaving just the address bits required by the tree routers. Each of these strips off more bits and passes the packet towards its respective TCP process, at which it eventually arrives with a routing value of zero.

Routes through the TCP routing tree may be in two states - "stream opening" and "stream established", corresponding to two groups of TCP states. When a TCP process starts to open a connection, it sends a message to the root router, indicating which IP addresses it is prepared to accept (in terms of IP addresses and TCP port numbers, which may contain don't-care bits). This information is stored in the root router's tables, together with the route value which specifies the return path to the original TCP process. Incoming TCP packets which match the address bit-map are routed to the relevant TCP process. As the TCP process proceeds with the establishment of a connection, it tells the root router about the precise values of the IP addresses and the TCP port numbers, and the root router table is updated. If two TCP processes attempt to open connections with similarly wild-carded addresses, the root router sends both sets of incoming packets to the first TCP process in its table - where the TCP three-way handshake will cause one set to be rejected. When the first connection is fully open, the refined address in the root routing table will cause other incoming open requests to be passed to the next TCP process interested in these packets, which will
in turn establish a proper connection and ask the root router to pass all relevant packets in its direction.

The other main function of the TCP router processes - both the tree routers and the root routers - is that of checksumming the TCP packets. The processes are instantiated with two rows of Boolean parameters, an element in one row for each of the incoming multiplexer input channels and an element in the other row for each of the outgoing demultiplexer output channels. The router performs a checksum calculation provided that the corresponding channel is marked appropriately and that the packet has not already been dealt with in a previous layer. This strategy allows for an asymmetric tree structure. Packets which arrive from the IP process and fail the checksum calculation are discarded. It should be noted that my code contained an error in this discarding mechanism which lay dormant and did not fail for over three years - a measure of the reliability of local area networks, such as Ethernet, compared with wide-area networks such as ISDN or X25 connections. This agrees well with observations of packet checksum error rates in [Braun91a]. Packets which pass the checksum test, or have a checksum installed before continuing towards the IP layer, are re-tagged to indicate that they have been processed. The last process in each side of the tree (the root router for outgoing packets and the highest tree router for incoming ones) must checksum all packets which have escaped attention at each other stage of the tree.

4.1.5. The TCP Process

The TCP process sits at the top of the hierarchy of protocol processes, providing an interface to the user's application process, as well as implementing the reliable stream connection to a similar destination process. It uses the underlying IP and physical layer processes to transmit data across the network.

The TCP protocol is specified as a state machine, with twelve states covering all eventualities from CLOSED through various stages to ESTABLISHED and then eventually back to CLOSED. Connections are opened on receipt of an open request from the local application process, and closed following an application closedown request, or a catastrophic error. In (primarily) the ESTABLISHED state, data is passed bidirectionally across the open network connection between user buffers which allows the application process to treat the network as a pair of First-In-First-Out buffers. Whilst, at each end of the connection, it might be possible to treat the transmit and receive operations in parallel, the transmission of user data and of acknowledgements to recently received network packets are best combined into the same outgoing packets, and these operations are currently performed sequentially in a single TCP process for each connection.

Occam is very suitable for the management of timeouts and for dealing with requests to carry out actions received from multiple places - all these requests may be dealt with using the ALT constructor. A large alternation construct forms the central part of the main TCP process, which loops continually, accepting incoming data from the application and the network, updating buffer
contents, sending data and acknowledgements to the network and handling the occasional timeout, retry or user command. The structure of the top level of this code (somewhat simplified) is shown in figure 34:

```occam
WHILE tcp.running
  ALT
    (tx.buffer.unsent > 0) & tx.req ? any
      ... send a packet to the IP layer, responding to a request from the transmit prompter
    from.ip ? tag; len16 :: ip.buffer
      ... deal with incoming segments from the IP layer
    rx.buffer.to.main ? received
      ... deal with notification that data has been consumed from receive buffer
    request.bytes ? request
      ... deal with request from user process for received data
    tx.buffer.to.main ? transmitted; pushed
      ... receive notification of data stored into transmit buffer
    transmit.control ? CASE
      ... deal with user commands
      ack.time.valid & timer ? AFTER ack.time
        ... send an acknowledgement if there has not been a data transmission recently
      retry.time.valid & timer ? AFTER retry.time
        ... reset send pointers and start retransmissions
      user.time.valid & timer ? AFTER user.time
        ... deal with user timeout
      tw.time.valid & timer ? AFTER tw.time
        ... deal with time-wait timeout
```

*Figure 34 - the occam code structure of the main loop in the TCP process*

Normally, only one of the "valid" conditions (on the last four timer inputs) will be true at any one time, somewhat reducing the perceived cost of this large ALT construct.

Since this alternation is performed once for each channel communication, it became necessary to write code to manage the buffers to the application process so that they were capable of accepting and transmitting data many bytes at a time, because the initial single-byte channel protocols at this interface proved incapable of providing satisfactory performance due to the set-up time of the ALT constructor. Even when user data is handled in maximum-length segments, the cost of processing each one through this large ALT construct dominates the performance of the TCP process.

Finally, the channel protocols used at the interface between the TCP process and the user process were chosen with care. One requirement of the overall software is to permit transparent bidirectional communication between occam processes running on separate processors, and this always requires careful design, even on raw networks of transputers, due to the need to implement flow control to prevent deadlock. In the TCP case, the flow control associated with the TCP buffers
had to be extended across the application interface as well, requiring a pair of channels for both the transmit and the receive sides of the stream. A further pair of channels was needed to pass user commands into the TCP module (such as initial configuration, stream opening and closedown commands) and to extract status and diagnostic information. An example of the use of these three pairs of channels is shown in section 4.1.6, below.

Figure 35 - the TCP process, its channel interfaces and shared buffers

The TCP process has four sets of shared buffers internally, two providing buffering for data passing to the lower protocol layers, and two acting as the main TCP receive and transmit buffers which are shared with the processes managing the TCP-to-application interface. Although they also share data with the main TCP window buffers, the two network-side buffers are relatively conventional.

One significant innovation is that the application-side transmit and receive buffers are not shared in the sense of multiple buffers whose pointers are passed around. Instead, they are made up from two large queues whose head and tail pointers are updated as the main TCP process adds or consumes data at one end and the I/O processes do the opposite at the other end. The communication between the processes is through two pairs of overwriting buffers which accumulate requests rather than passing them on individually. This scheme is especially important to reduce the message rate flowing into the main TCP alternation process.

Two different strategies for transmitting towards the IP layer have been tried.

Initially, whenever the transmit process reported the arrival of user data to the main TCP process, all of it was broken up into segments and transmitted as a continuous stream of network packets before the main ALT construct was re-entered. Additionally, this reporting was delayed (by the accumulating buffers described above) until sufficient traffic had built up for such a transmission,
or until the application signalled (via a push request) that a transmission was essential. The output process in figure 35 is just used as a FIFO buffer in this case, and the reverse request channel is ignored. This method minimises the number of actions which the main TCP process has to perform, but leads to a build-up of incoming messages and acknowledgements from the IP process, which can cause the underlying buffers to become full, and for paths through the TCP router hierarchy to block unnecessarily.

A second technique which has been tried is to run the output process as a prompter - requesting each packet from the main TCP process and then passing it on towards the IP layer. It is, of course, configured as a double buffer, so that the segment communications on both sides of it can proceed in parallel. The main TCP process now receives frequent requests from the output process, and responds immediately by sending a single TCP segment when one is available, or quietly records the readiness of the output process if not. The sending algorithm attempts to construct maximum-length packets, but only sends one at a time in contrast to the previous scheme. Attention must be paid to non-data transmissions (control packets, acknowledgements and packets in transit from the user application direct to lower-level protocol-processing processes), but this extra request/response code may otherwise be added mechanically.

Surprisingly, the performance of the two different strategies is virtually identical when carrying high volumes of unidirectional traffic, and the second strategy is only a few percent better in the bidirectional case. The latter, however, is far less sensitive to the provision of buffering in the underlying layers of protocol processing, and is the preferred technique. These buffering issues are discussed in section 4.3, below.

4.1.6. The Application-to-TCP Interface

Another important aspect of the TCP process design which contributes to its speed is the user interface. It would be possible to provide a TCP-to-application interface which used a pair of channels, one for the application process to issue commands and to send data to the TCP process, and one to receive data, acknowledgements and status information from the TCP process. This would correspond to the style of system call interface provided by most operating systems for filestore and socket access - supporting separate open, close, read and write procedures which all operate synchronously. In a parallel programming environment, however, it is more natural - and often easier - to design processes which perform reads and writes to their neighbours asynchronously and in parallel. In the case of networked applications, the neighbours that we refer to are, of course, the processes to which they are connected elsewhere on the network. Hence a TCP-to-application interface which supports parallel reads, writes and control is necessary for maximum performance and generality. If need be, this interface may be connected to less-demanding applications via a parallel buffer process which provides the conversion to the simpler form of socket interface mentioned above.
After the experience of writing a number of server and client applications, the TCP interface finally selected has three pairs of channels separating the TCP process from the application processes. It is illustrated, above, in figure 35. The control and status pair are used to pass commands to the TCP process and to receive asynchronous status information from the TCP process, some in response to previously-issued commands and some gratuitously provided for diagnostic, performance monitoring and connection status purposes. By using a separate pair of channels for these functions, there is never any need to worry about the control channel becoming blocked if the network fails and outgoing data buffers up and blocks the transmit data channel.

The receive data pair operates on a conventional request / response basis, with a minor enhancement. Normally, the application requests a number of bytes of data from the TCP process. This request blocks either until the request is satisfied in full, or until a push flag is received in the incoming data stream, in which case the partially-full buffer is returned immediately and the remainder of the request is cancelled. If a zero-length request is made, regardless of whether the previous one has returned, a zero-length response will be generated, indicating that any outstanding request has been cancelled. The application must be prepared to receive a non-zero-length response in this circumstance, however, since this may have been transmitted at the same time as the zero-length request was sent. In this case, a second (zero-length) response follows. The need for this termination mechanism is often found when data embedded in one packet indicates the end of the stream and double input buffering is being used; the buffer process must be closed down after it has issued another request.

The transmit pair comprises a blocking transmit channel and a request channel. In most circumstances, it is easiest to write an application process which only sends data to the TCP process after the TCP process requests it - thus allowing a single application process to be built up around an alternation statement which accepts inputs (including the output request) and only performs actions in response to them. Occasionally, however, it is possible to obtain better throughput by sinking requests from the TCP process into a process which runs in parallel with the application and to allow the application's data output be blocked by the TCP process. Regardless of whether the request channel is used properly, the TCP process will always be able to accept 2000 bytes of data when it reads from the application process.

TCP is defined always to empty the transmit channel for each connection - eventually - and synchronisation between channels in the application process must be performed to ensure that this is done successfully, regardless of whether the connection is closed normally or aborted. To this end, it is necessary to transmit a zero-length message from the application process to the TCP process, immediately before issuing the closedown command. The TCP process waits until it has processed this message before beginning the connection closedown, in order to ensure that all transmit buffers have been flushed.
4.1.7. Tailoring the TCP Buffering to a Pipeline

Other performance enhancement schemes investigated have been related to the longer latency paths inherent in a pipelined implementation. Since, in a busy system, packets will be queued behind others in the incoming and outgoing pipelines, many may be in transit at once. This circumstance will be especially noticeable if there is a constriction at the Ethernet itself (see section 4.1.1.2 for remarks about the communication performance of the IMS B431 Ethernet TRAM).

The effect of a long latency is that a single TCP process will only be able to operate continuously if it has enough data to transmit at all times, and if the incoming protocol acknowledgements allow it to do so. The TCP sliding window scheme allows for transmit and receive buffers of up to 64 kbytes, and full benefit must be obtained from these if a high performance is required. Unfortunately, many implementations of TCP treat the 16-bit window header field as a signed quantity, and it is therefore necessary to restrict the advertised window sizes to 32 kbytes to avoid compatibility problems. This corresponds to the user data in about 22 maximum-sized TCP packets on Ethernet, and with several packets distributed across each of the Ethernet controller, IP, TCP router and TCP processes in both the transmit and the receive pipelines, it is clear that the receiving TCP must advertise all its free buffer space frequently in order that the notification is received and acted upon before the transmitting TCP process goes idle. Ideally (as is also found for high-speed, long latency satellite links), a larger window announcement field and larger buffers should be provided in future protocols designed with high performance in mind - see, for example, [Jacobson92].

Conversely, much buffer memory can be saved when a lower performance is tolerable or when the higher-level protocol in use requires end-to-end acknowledgements on a regular basis (e.g. Iserver), and in this case attention must be paid to latency minimisation instead.

4.1.8. The UDP Process

The UDP process supplies an interface to the unreliable datagram service. This service corresponds directly with that provided by the IP layer of the TCP/IP protocol family. A UDP message is transmitted as a single IP packet, checksummed and addressed to unique or broadcast IP addresses, and with 16-bit port identifiers on both local and remote hosts [Postel80 - RFC768]. There is no flow control or acknowledgement mechanism (unless the receiver should send another UDP packet in the opposite direction as part of a higher-level application protocol). IP does not guarantee that a UDP packet will be delivered intact, or that several UDP packets will be delivered in the same order. UDP is therefore only suitable for information exchanges where occasional packet loss is irrelevant, or can be coped with by a simple timeout and retry mechanism.
Since IP packets may be fragmented before transmission on the underlying network, packet loss will cause the re-assembly of such fragments to fail. In this implementation, therefore, the UDP packet length has been restricted to 1450 bytes, which allows UDP datagrams to be transmitted in one Ethernet packet, even after the IP and Ethernet headers have been added.

The UDP process (figure 36) is connected to the IP process via the TCP router tree of processes. After initialisation, it is composed from two completely separate processes, one for transmit and control operations and one for receive operations. UDP packets may be transmitted at any time, along a single blocking channel which accepts complete packets. This channel may also be used to nominate IP address pairs and UDP port pairs which are used by the root router process to select incoming UDP packets for each UDP process. Wildcard fields may be included in these specifications.

A two-channel interface is provided to the incoming UDP stream. The receiving UDP process provides an overwriting FIFO buffer of recently-received packets which satisfy the selection criteria. Packets may be extracted from this buffer by sending a dummy value along a request channel. Each UDP process in any particular configuration provides its own overwriting buffer, thus ensuring that none will block traffic being fanned-out in the TCP demultiplexer tree. Notice that UDP packets may be copied by the routing tree to several UDP processes, which cannot happen in the case of TCP packets.

4.1.9. Conformance to the TCP/IP Specifications

[RFC 1122 - Braden89a] provides check-lists of mandatory, desirable, optional, undesirable and forbidden features of all the basic TCP/IP protocols. The code described in this thesis satisfies most of these points, but is structured in a manner which makes some impossible, mainly because the transputers are not running a conventional operating system. Notable divergences from the RFC 1122 requirements are:

- The IP process currently does not attempt to detect failed gateway stations and then to re-route messages via another gateway. This has not been significant in a tree-structured Ethernet network environment where there is only one gateway on each sub-net.
• No type-of-service (TOS) or precedence checks are performed in any of the protocol processes. This does not appear to be essential to a local Ethernet implementation, since no non-zero values of these parameters having been recorded over several years of operation. Addition of these features will involve no structural changes, and only a few lines of code.

• No multihoming code has been implemented at the IP level, since the current code is designed for simple client and server operations with just one network interface. The IP layer does not provide any gateway services, for the same reason. It would be possible to link together two LANCE processes, talking to two Ethernets, by two IP processes and a new gateway process to perform inter-network routing.

• TCP/IP packets are limited to 1500 bytes in length, so that they will fit into Ethernet frames without fragmentation. Conveniently, this value can also be used as the declared length of the packet buffers in the processes along the pipeline. This imposes a practical maximum size of 1450 bytes for TCP packets, with the rest given over to the IP header plus options. Since all incoming single-fragment Ethernet packets will fit within 1500 bytes, they will all be processed successfully. Excessively-long reassembled packets will be discarded, but in practice this is not an issue, because the TCP process negotiates a maximum segment size for each connection, and this is used to limit the overall size of both incoming and outgoing packets.

• The socket-style TCP to user-process interface is re-implemented in a fashion more suited to a message-passing environment - see section 4.1.6 for full details. The new scheme has considerably reduced latency and provides better asynchronous control than the one specified, but otherwise has a similar functionality.

In addition, some relatively minor features which are specified as compulsory, but which may be also disabled at run-time by an application-level parameter or command, have not been implemented.

There is, of course, no Task Control Block (or TCB) for each connection, as specified in [RFC793 - Postel81c]. The information contained in this table is distributed amongst the tables in the TCP routing processes, as well as in the local data of the individual TCP processes themselves.

4.1.10. The Strategy for Deadlock Avoidance

Part of the design work for any parallel computer project should be a justification of its deadlock freedom. In the case of this TCP/IP software, the pipelined structure made my task relatively straightforward, but all communication between the incoming and outgoing processes at the network and user application ends had to be treated with care. Consider the processes and channels in figure 37:
The method that will be used to prove that the process structure is free from deadlock is to traverse all of the cycles which may be seen in the diagram, checking for when each of the channels commits to communicating. If it is possible for every channel in the whole of any cycle to become committed at once, the program will deadlock. This strategy is similar to the underlying principles in [Roscoe87] and is a simple case of the interconnected client-server model presented in [Welch93]. Whilst the discussion is rather involved, requiring discussion in turn of each of the numbered channels in figure 37, the summary at the end of this section is quite straightforward.

Ethernet is a common-bus network on which only one packet travels at any time. When the network is busy, traffic will be queued for transmission by the Ethernet hardware interface. Outgoing packets will be delivered to stations on the network dependent on their embedded addressing information. Stations must be prepared to receive traffic, even when wishing to transmit themselves. Eventually, each station will be able to send packets onto the network (thanks to its low-level contention management strategy), and the transmit channel (1) will thus never be blocked for extended periods. During the time when it is blocked, however, many packets may be received on the Ethernet receive channel (2). FIFO buffering within the LANCE process, mainly associated with the controller's memory-mapped hardware buffering scheme, will relieve the short-term fluctuations in packet delivery. These buffers are only small by comparison with the number of packets which may be crossing the network every second, so it is likely that channels (3) and (4) will block occasionally. Channel (3) will always unblock in due course, when more of our traffic is allowed onto the network, and it is our responsibility to ensure that no rate of message delivery on channel (4) can cause deadlock problems to the rest of our architecture.

Since channel (3) will not block indefinitely, the IP transmitter process (IP.tx) may be allowed to transmit along it at will, and will itself never be deadlocked. It may, however, occasionally need to send configuration information to the IP receiver process (IP.rx). Each time that any of the items needs to be sent along this channel (5), they are all packaged together and sent as one unit. This
package is passed through an overwriting buffer which ensures that the most recent values always arrive and that the IP transmitter process is never blocked by this channel.

In normal circumstances, the IP transmitter process will receive frames from the TCP routing hierarchy (here represented by just the root router (TCP.rx.router & TCP.tx.router)) along channel (6). The root router transmitter sends a single message along channel (7) to the root router receiver each time that a TCP process opens or closes a connection, and each time that the similarly-placed UDP process nominates further packets for it to inspect. Only one of these messages needs to be passed at any time for each active stream, and therefore an overwriting buffer process with a separate slot for each TCP process is provided on channel (7). Since only one TCP connection is maintained by any TCP process, one slot is sufficient for each TCP connection. On the other hand, a UDP process may wish to nominate several addresses from which it wishes to monitor incoming datagrams, and the overwriting capability guarantees that channel (7) is not flooded. After sending a request to the root router, an application driving the UDP process must wait for a reply before it may send another; this limits the TCP root router buffering requirement to one slot for UDP activities, too. Maintenance of this behaviour is the responsibility of the user, but if the UDP process were modified to discard excessive UDP command traffic, it would only be doing the task that the overwriting buffer slot currently performs, and communication between the parallel transmit and receive pipelines in the UDP process would require complicated orchestration. Our discussion of channels (6) and (7) justifies that the root router transmitter will not provoke deadlock. The TCP tree routers which may be installed in addition to the root router are pure pipelines and do not communicate between their transmit and receive sections. Therefore they add to buffering in the pipelines but cannot contribute to any deadlock.

The TCP process is especially complicated, since it joins the receive and transmit pipelines, and so it must be treated with care. Remember, however, that channel (8) cannot block forever. This permits the TCP process to transmit whenever it likes onto channel (8). The earlier transmission policy (outlined in section 4.1.5), which sent the whole of its transmit buffer in one unbroken stream following receipt of data from the user or after the retry timer expired, would tend to saturate the transmit pipeline, and although it may have been slowed by blocked messages, it would utilise the Ethernet effectively. It would also prevent the main TCP ALT body from responding to incoming packets for long intervals, which is a non-optimal but adequate behaviour. The newer transmission policy (which only sends a few IP packets before returning to the main ALT loop) interleaves transmissions with receptions in a fairer manner, and provides a similar overall performance. Since the transmit and receive channel pairs (two of the pairs labelled (9)) which connect the TCP process to the user application process simply exchange user data with flow control tokens (as client-server pairs), and operate on the shared TCP data buffers, they will never deadlock (if the client-server rules are applied properly!) and serve to isolate the TCP process hierarchy from the user. Similarly, the TCP status channel to the user process must always be read by a separate parallel process in the user application which must have overwriting properties. All
commands received from the sixth user interface channel will either be consumed in the TCP process or will be converted to control packets which are transmitted along channel (8). We have already established that this channel will eventually consume these packets.

Next, we come to the TCP receive channel (10) which is connected to one of the input guards in the main TCP ALT loop. Messages on this channel will be consumed in the ALT’s usual non-deterministic fashion whenever they become available, although the choice of the TCP transmit algorithm may cause them to be blocked for a while. Received packets may be data (which is stored directly into the receive buffer) or acknowledgements. Received data will provoke the transmission of user data, or an empty acknowledgement packet if there is nothing else to transmit. Received acknowledgements update the transmit window but must not cause an acknowledgement to be sent out - acks-of-acks are a good way to drown the Ethernet with thousands of empty packets per second. All we know about the incoming data stream is that, at any time, only as much data may be in the stream as we have advertised space in the TCP receive window (up to 32 kbytes), but we do not know in how many packets it may have been transmitted. Channel (10) will, therefore, never become blocked indefinitely, but may delay traffic on a short-term basis.

The receive side of the UDP process is configured with an overwriting buffer which therefore cannot block its receive channel (also (10)). The behaviour of this overwriting buffer is consistent with the unreliable delivery guarantees of the UDP datagram protocol.

The TCP receive router tree of processes delivers TCP and UDP packets to their respective processes. We have seen that each output channel (represented by (10) in figure 37) will not deadlock, but may block if its TCP process is not fast enough. This leads to the possibility that some paths through the demultiplexer tree may become blocked, and not allow messages through to other TCP or UDP processes, on a temporary basis. If this is unacceptable, buffering must be provided upstream - at (10) - to alleviate the situation. Without conglomerating small packets and deleting superseded acknowledgement packets (which are otherwise empty), it is impossible to predict the maximum size of buffer which is needed to avoid any channel interference. It is far better to ensure that all TCP processing is performed promptly, so that any blocking in the tree is short-lived. There is no path from the TCP root router receive process back to the TCP root router transmit process, so the previous discussion is enough to decide that this process is deadlock free and that channel (11) does not block indefinitely.

The IP receiver process has two outputs, channel (11) which has been examined above, and channel (12) which connects it to the IP transmitter process. The latter passes several important messages at start-up time (such as the station’s IP address and the length of the transmit buffers in the LANCE process). These messages are allowed to propagate along the channel in a blocking fashion, before all the other TCP and router processes start up and the channel takes on its steady-state role. From this time, the channel passes received control packets (mainly ICMP and ARP messages which update the routing tables) to the IP transmitter process. It also constructs ICMP
and ARP replies in response to other stations' requests, and sends these to the IP transmitter process for transmission onto the network. Since the IP transmitter channel (3) never becomes blocked for long, nor will channel (12).

At last, we return to the Ethernet control process. This will always be able to deliver packets to the IP process along channel (4), as a conclusion to the argument above, although it may be blocked occasionally. Our only concern, therefore, is to code this process with sufficient parallelism to ensure that packets may continue to be transmitted along channels (3) and (1) even while the receive section is blocked by channel (4). This is simply ensured by discarding incoming Ethernet frames when the LANCE FIFO buffer fills up - indeed, this is the normal hardware behaviour. This is essentially an overwriting buffer. Although it throws away the most recently-received packet when it overflows, it still breaks any opportunity for a cycle of committed communications to develop throughout the whole loop of processes. We might choose to install extra buffering along the receive pipeline to reduce the likelihood of packets being destroyed unnecessarily. It is also a good idea to install such buffers at each interface between transputers in the pipelines, and to run them at high priority, so that we may maximise the parallelism of protocol processing and transputer link communications, and thereby optimise throughput.

In summary, this section has outlined the strategy which has been adopted to produce a deadlock-free process structure. It demonstrates that the LANCE transmitter will never be blocked for long whilst transmitting onto the network, and that the IP.tx process, the TCP.tx router processes, the TCP and UDP processes, the TCP.rx router processes, the IP.rx process and the LANCE receiver process all act as clients, in turn, to the previous process in the list. They are all deadlock-free for the reasons stated in [Welch93]. The cross-links on channels (5), (12) and (7) in figure 37 all contain overwriting buffers, and thus act as non-linked servers, breaking any deadlock cycles through these paths. The overwriting behaviour of each of these buffers has been shown to operate in sympathy with the traffic carried - nothing vital is thrown away. The whole process scheme, as pictured, is therefore deadlock-free.

Care has also been taken, within the major processes described, to ensure that no parallel sub-processes can deadlock. In these cases, the programming paradigms used are sufficiently common and well-known, such as groups of I/O-PAR processes in the router multiplexers and demultiplexers, and overwriting (or accumulating) buffers in the TCP process, that a full discussion of the deadlock properties of each is unnecessary here.

4.2. Techniques used to Maximise TCP/IP Throughput

The data - as opposed to control - information within each TCP packet should only be manipulated for two reasons. Firstly, it has to be moved along the pipeline at each stage - using input and output instructions which may invoke DMA communication between transputers or block-move copies between the workspaces of processes running on the same transputer. This activity is unavoidable,
but moving 1 Mbyte per second along a pipeline should not be difficult for transputers with links capable of more than 1.7 Mbytes/second of sustained unidirectional throughput, and with an external memory bandwidth of at least 12 Mbytes/second. Delays in starting successive transmissions will have an effect on the message rate, however, and it is these which must be minimised to keep the pipeline flowing effectively.

Secondly, the whole of each TCP packet is checksummed once at each end of a connection. The same checksum routines are also used to sum the IP headers, which are much smaller than a typical TCP data packet. The original, simplistic, implementation of the checksumming routine was only capable of summing 525 kbytes/second on a 20 MHz transputer - resulting in an overall TCP transfer rate of under 200 kbytes/second. This corresponds to line "checksum1" on the graph in figure 38 below. All the checksum measurements in this section were performed on a 20 MHz T800 transputer, with all code in single-cycle on-chip memory and all the data packets being checksummed stored in slow 5-cycle off-chip memory.

![Figure 38 - checksum algorithm - throughput for all Ethernet packet sizes](image_url)

Optimisations were then applied to this basic algorithm, using the applicable recommendations in [Braden88 - RFC1071] and [Clark82 - RFC817]. The incremental updating of checksums described in [Mallory90 - RFC1141] and [Rijsinghani94 - RFC1624] is not relevant to this implementation, since it does not modify already-checksummed packets as a gateway node would.
Careful attention to the associative nature of the checksum algorithm, by which neighbouring bytes can be taken together and summed as 32-bit words, nearly doubled the throughput, but also exposed the overhead of adjusting the the word alignment of the result after summing these 32-bit values. Attention to the placement of the TCP and IP data in the buffer reduced the adjustment effort, giving a new rate of nearly 1.9 Mbytes/sec ("checksum2").

Rather than looping around the summation code for each 32-bit word encountered, the loop was then unrolled so that 8 words were added each time the loop code was executed, and a small loop at the end was used to sum the few remaining words. Just summing 8 words at a time was considered the correct compromise (saving 7 loop tests in every 8) since 8 words (or 32 bytes) is only slightly smaller than the 36-byte size of most IP headers and exactly right for zero-length TCP segments (empty acknowledgement packets). This allowed many common small checksums to be calculated using the optimised routine rather than falling completely into the small final-processing loop. This loop unrolling improved the summing rate to 3.4 Mbytes/second ("checksum3").

Only then was assembler code resorted to, eliminating an unnecessary load and store operation for each word summed, and yielding a final throughput of 4.4 Mbytes per second ("checksum4"). Notice that the assembler coding - just four lines of sequential code repeated eight times - contributed to no more than 23% of the eventual throughput. It is expected that a better optimising compiler could provide this improvement directly. Also notice how flat the curves are for packet sizes above 500 bytes - maintaining all packets at their maximum size of 1500 bytes will not make a large improvement to the checksumming performance over those only a third of that size.

It is also instructive to examine the checksumming throughput for the small packet sizes. The graph in figure 39 expands the one in figure 38 for small packet sizes, and illustrates the efficient summing of packets which are a multiple of 32 bytes in length, and the impact of summing the remainder when loop unrolling is used. Notice, also, that the unrolled loop algorithms are substantially faster, even for small packets of 32 or 36 bytes.

The difference in performance obtainable by using 3-cycle off-chip memory to store the packets being checksummed rather than 5-cycle memory may be calculated directly from the figures above. A rate of 4.4 Mbytes/sec corresponds to 909 ns per word read, and a saving of 100 ns would yield 4.95 Mbytes/sec, which is closely verified by measurement (5.02 Mbytes/sec).

Another conventional programming technique that was tried in the quest for maximal TCP/IP performance was to implement many of the function calls as INLINE FUNCTIONs in the D7205 occam implementation. Whilst this was partially successful, the increased data access times caused by having more local variables in each workspace tended to decrease the overall performance, demonstrating that this facility must be used with care.
No significant benefits were achieved from implementing complex routing table look-up algorithms in the IP and the TCP router processes, due to their increased start-up costs which would only offset their lower running costs if the tables were larger than our small embedded systems usually require.

4.3. Code Development, Testing Strategy and Verification

At the design stage, it was decided that all of the messages passed between the various TCP and IP processes should be accessible for diagnostic purposes. Unique BYTE protocol tags were incorporated throughout (rather than allowing occam to set tag values within a tagged protocol automatically), so that particular messages could be identified using well-known tags. They could then either be acted upon or passed through at each layered protocol process. This permits diagnostic traces of lower-level protocols to be extracted, and messages to be passed in from the user interface for processing just at the lower levels of the protocol stack. Incorporating such communication at a later stage would have been extremely difficult.

It is interesting to note that the code development strategy was considerably simplified by the choice of this multi-process, pipelined structure, and by the use of occam. Protocols such as TCP are peer protocols, which means that, from higher-level protocol layers, two TCP protocol
processes appear to communicate directly, although usually they rely on lower-layer protocols such as IP and the network hardware to exchange information. It is therefore possible to examine each individual protocol process separately, and to join two such processes back-to-back to test them, as shown in figure 40. In this situation, it is necessary to install two overwriting buffer and filter processes between them to maintain the conditions for deadlock-freedom introduced above. These small buffers may have to be extended into longer FIFO buffers to avoid packet loss - see below. The filter processes must remove packets which would normally progress to lower-layer processes (such as to the LANCE controller process) and must translate the tags and protocols of transmitted packets to simulate those of received packets. Suitable application processes attached to each of the TCP processes may then be provided to exercise the whole configuration. The same occam channel protocols may be used in this test environment as will be used for future live running.

Such testing may be performed both from the top and from the bottom of the protocol stack, allowing TCP, TCP plus routing, IP and other combinations to be tested in isolation, away from the Ethernet. Testing pairs of processes back-to-back simplifies error detection, because it is unlikely that erroneous code will also contain the inverse error in the path running in the opposite direction. Errors in the interpretation of the protocol specifications are less likely to be detected by this method, however, since the same fault will probably be replicated in the transmitting and receiving processes.

The protection provided by the occam channel protocols as well as by occam’s type, usage and alias checking makes combining these processes straightforward and reliable. It was notable that bidirectional data was successfully transferred to a C program running on a Unix host at the very first attempt following assembly of the complete set of processes.

A persistent problem which took a long time to identify, and even longer to characterise fully, was that of instability in the protocol when two TCP processes were joined back-to-back as in figure 40 above. In principle, it is possible to deduce how many TCP segments might have to be buffered in
the filter processes at any time. This number depends on the transmission strategy, the number of acknowledgements being produced in response to data passing in the opposite direction, and the size of the TCP transmit and receive buffers (and thus advertised window sizes).

Unfortunately, if fewer FIFO buffer elements are provided than necessary, the overwriting buffer in the filter processes will drop packets, and the TCP process will eventually re-transmit them. The retransmission algorithm provided in my TCP process is very simple, since packet loss is rarely seen on real local area networks. When the retry timer expires, it sets the transmit queue pointers back to the first unacknowledged data, and the packet transmission routine then re-sends the whole buffer. More efficient algorithms are suggested for low-reliability links, but these usually rely on re-transmitting just the packet which has not been acknowledged, and the storage implications of this are unattractive for a small embedded-system implementation. A re-transmission of the whole transmit buffer therefore floods the overwriting buffer in the test harness, and provokes a stream of acknowledgements which can make the matter worse. If the overwriting buffer destroys its oldest entry when it fills up, the first packet to be retried (the one which was not acknowledged the first time) is destroyed again, and the situation can become stable, causing no further traffic to flow successfully. If the youngest entry in the buffer is overwritten, only a few packets are successfully passed to the receiving TCP process before one is lost, and this causes a delay until the retry algorithm re-sends it. This situation will also tend to be stable, and results in reduced throughput. Both situations are a result of providing ridiculously small FIFOs (of the order of 1 to 4 packets long), although performance reductions through packet loss can persist until the FIFOs are able to accommodate the complete contents of the TCP transmit and receive buffers.

Once identified, the problem was resolved by converting to the TCP transmission scheme which uses a prompter process on its output and only generates segments on request. Ideally, this would be operated together with filter processes which do not contain overwriting buffers, but these are still needed to maintain the deadlock-freedom conditions under which the TCP process was designed. However, by prioritising the main TCP alternation process (in favour of packet reception) and by ensuring that overwriting is only performed as a last resort (after a timeout to allow further reception operations), a reliable test environment may be produced, which only occasionally destroys segments in transit.

4.4. TCP/IP Performance Evaluation

The double pipeline approach to processing TCP/IP packets has been demonstrated to support data rates, between three transputers and a Sun IPC workstation, across Ethernet, of up to 950 kbytes per second [Peel92b]. I believe that this is three times as fast as commercially available products such as the INMOS B300 can achieve. More detailed performance measurements have also been made, which are reported in sections 4.4.5 and 4.4.6 of this thesis.
Most of the measurements have been obtained by incorporating the major pipeline process elements into simple test-bed applications and then pumping large numbers of standardised packets through them. These packets were constructed to exercise just the commonly-executed paths in the pipeline, not the ones used for connection opening and closing or for error recovery. The sizes of the packets used at each stage in the pipeline mirrored the changes as additional headers were added or removed, depending on the direction of travel. In most cases, the process elements were exercised while installed on the middle transputer in a pipeline of three such processors. As well as packet generation and consumption, the processors at the ends of the pipeline were responsible for deriving timings for each run; the results are expressed as a bandwidth (or throughput rate) for the pipeline elements in question. Incorporating each process under test into an active pipeline enabled its performance to be measured in an appropriate context.

In the more complex case of the TCP process, two instances of this were joined back-to-back by long FIFO buffers, and a connection opened, heavily exercised and then closed. Again, the TCP processes were run on separate transputers and the intermediate buffers on a third. Due to the tight coupling of the transmitting and receiving TCP processes, it was impractical to evaluate their transmit and receive performances separately, and the figures presented are a measure of the overall effect of high-speed TCP traffic.

The transputers used in these measurements were 20 MHz IMS T800-Cs and IMS T800-Ds, with 5-cycle external memory, except for those indicated. Improvements may be seen when increasing their clock rate and when reducing the number of cycles per external memory access from 5 to 4 or 3, but these have not been plotted in most cases due to the difficulties of incorporating available boards with increased specifications into the measurement configurations. Transputer links running at 20 Mbit/sec have been used throughout (except for some of the T2 DMA tests in section 4.1.1.2), and link acknowledgements were properly overlapped on all devices.

4.4.1. Throughput of the Test Harnesses

The first measurements presented are of the test harnesses themselves - they must not impose appreciable delays on communications from the processes under test if subsequent figures are to be meaningful.

Three main processes were placed on three transputers, as shown in figure 41. At the head of the pipeline, the producer process simply sent messages to the consumer process, using various occam protocols, and the throughput was measured. In-between, Transputer 2 ran a double buffer process to act as a place-filler for the real process which would be installed in this position later. On each transputer, the process code was run entirely in fast on-chip memory, and the pipeline data buffers were placed in the much slower off-chip memory.
The three major processes were run at low priority, and were joined by two pairs of double buffers, running at high priority. High priority buffers are required to maximise the throughput of the transputer links, as recommended in [INMOS89b], chapter 14; their exclusion reduces throughput by about 10%. Figure 43 shows the data rates obtained for messages of various lengths, when the pipeline channels were used to carry various protocols which correspond to those in the TCP/IP implementation. The top graph shows that the \texttt{INT16::[BYTE}} protocol achieves a rate of 1.74 Mbytes/sec, which is the maximum throughput given in the Transputer Databook [INMOS89a] for this processor with 20 Mbit/sec links. For smaller packet sizes, the rate tails off as the effects of transferring the packet length (in the \texttt{INT16} field) and of converting it to a 32-bit quantity became a larger part of the overall task. Looping instructions within all the processes were also exposed as a significant overhead when running with reduced packet sizes – unrolling was not used.

Moving downwards, the \texttt{BYTE; INT16::[BYTES}} protocol, which mirrors that used by the IP processes, achieves virtually the same maximum throughput for large packet sizes, as does the \texttt{INT32; BYTE; INT16::[BYTES}; \texttt{INT16::[BYTES}} protocol, which reflects that used by the TCP and TCP router processes. The next two lines show the effect of receiving two \texttt{INT16::[BYTES} protocol elements into a single memory array, and passing the combined contents on in a single message with protocol \texttt{INT16::[BYTES} - the necessary behaviour of a filter process which connects a TCP transmitter process to a TCP receiver process. This concatenation is performed by reading the second row of bytes into the correct position in a buffer which already contains the first row, so no data copying is required, as shown in figure 42:

\begin{verbatim}
SEQ
   prod.to.buff ? route; byte; len1 :: buffer;
   len2 :: [buffer FROM (INT len1) FOR (bufsiz - (INT len1))]
   len := len1 + len2
\end{verbatim}

\textit{Figure 42 - concatenating two row communications into a single buffer}
The precise semantics of this operation only appear to be defined by the early INMOS compiler implementations, according to Geraint Jones. Apart from the slightly higher per-message overheads, which cause a loss of throughput for smaller packet sizes, there is very little degradation in throughput for this additional operation.

Having established the performance of the communication links, and of the protocol conversions required, we can examine the throughput of the TCP-to-TCP and IP-to-IP filter processes. The former is pictured in figure 45, and the latter in figure 61. Each filter process incorporates an overwriting buffer as well as a FIFO queue. Incorporating the FIFO decouples the trees of protocol-processing processes on either side, and the overwriting buffer ensures deadlock freedom where necessary. The overwriting buffers were disabled in the throughput tests in this section, to ensure that all the packets transmitted by the producer process arrived intact at the consumer process, and that the measurements were not upset by destroyed packets.

The top trace of figure 44 shows the throughput of the IP-to-IP filter. This simple FIFO does not need to do anything to convert incoming packets of protocol TX.PROTO to protocol RX.PROTO - they both carry messages of type BYTE; INT16::[1]BYTE. Its main task is to convert LANCE.TRANSMIT tags to indicate RX.DATA - pretty trivial. Not surprisingly, throughput is only a fraction lower than the corresponding transmission through a plain double buffer.
Due to a bug in the D7205 compiler used, the in-situ buffer concatenation method described above does not work in the TCP-to-TCP buffer, so actual copying of the second row of bytes onto the end of the first row has to be performed. The second trace illustrates the effect of copying no bytes, and shows little degradation at large packet sizes and a larger reduction than the previous case for smaller packets. This degradation is due to the FIFO code passing the extra empty field, and represents a per-packet overhead. For packet sizes greater than 1200 bytes, the linear section at the top end is due to saturation of the communications paths, whilst the remaining part of the curve shows that the processor is fully occupied running the FIFO algorithm.

The lowest trace shows the effect of copying all the bytes, and adds a per-byte cost to that of the FIFO, above. For large packets, this cost is highest, whilst for small packets the extra overhead is small - but so is the data rate. These two TCP-to-TCP traces show the range of throughputs which real TCP processes can expect to see when connected together by the TCP-to-TCP filter process. In practice, the effect of the data copying has been reduced in all my TCP tests by performing the concatenation, using the method of figure 42, in one of the high-priority buffer processes between the output of the TCP process and the input of the TCP-to-TCP filter process. Thus, for TCP packets of nearly 1500 bytes, we have a filter capacity of 1.65 Mbytes/sec, only 6% down from the raw link capability of 1.74 Mbytes/sec.
4.4.2. Throughput of the TCP Process

We can exercise the TCP process by installing copies in place of the producer and consumer processes in figure 41, above, to yield figure 45. The TCP processes are driven by small application processes which simply transmit messages and receive messages, respectively. Varying the sizes of all the interesting variables enables us to watch the effects in the measurements below.

![Figure 45 - measuring the TCP process performance](image)

Firstly, messages of a fixed (but reasonably large) size were sent from the user application process to the transmitting TCP process, and the size of the buffer into which we received data in the receiving application was varied. Here, 1400-byte messages were sent, and a 1400-byte TCP segment size was used to carry them across the simulated network. Additionally, TCP send and receive window buffers of 30 000-bytes were used. In principle, we would expect that the largest application receive buffer should yield the highest throughput, because the per-message overheads would be minimised.

This was indeed the case. The bottom trace in figure 46 represents the unidirectional transfer rate when the processes illustrated are run on three transputers, and shows a peak at 1409 kbytes/second. The curve is reasonably flat - user buffers of 400 bytes or more are capable of sustaining the Ethernet transfer rate of 1.2 Mbytes/sec. Remember that the test application simply received data into a buffer in memory and then ignored it, however; real applications will have to access the buffer at least once more in order to use the received data.

The top trace in figure 46 shows the total throughput when both sides of a connection were run simultaneously - each stream nearly halved in throughput when two were run together. The maximum bidirectional rate of 1451 kbytes/second is only a little more than the unidirectional rate, and shows that the memory of the transputers was fully utilised. With reduced performance logging, bidirectional rates of up to 1490 kbytes/second have been seen in this configuration, but all disturbances to the code have an effect on throughput at these extreme rates. The bidirectional throughput is extremely insensitive to varying the receive buffer size - it hardly degrades at all until the buffer size falls to 200 bytes. This indicates that the TCP acknowledgement packets, generated as each data packet was removed from the TCP buffer, were better utilised in the return direction and that the links and the TCP-to-TCP process were the limiting factor.
Next, the effect of varying the application's transmit buffer size was examined, in both the unidirectional and the bidirectional cases. All other parameters were maintained as above, and the application receive buffer was fixed at 1400 bytes - the size which yielded the peak throughput in the previous measurements.

This time, figure 47 shows that the bidirectional throughput peaked at over 1430 kbytes/second for application transmit packet sizes of 400 bytes upwards, with maxima of 1460 kbytes/second for both 400-byte and 1400-byte packets. This traffic was transmitted across the simulated network in packets of data size 1400 bytes, plus headers of 16 bytes, making a total of 1031 packets per second or 1460 kbytes/second. It is safe to assume that all these packets carried acknowledgements for free, and that no small acknowledgement packets were generated by the TCP processes. Referring to the plot of the TCP-to-TCP process throughput in figure 43, this shows that the test harness was capable of 1630 kbytes/second at these packet sizes, if all the data were to be included in the first of the two data fields in the TCP_TX transmission protocol. This indeed would be so, following its concatenation in one of the high-priority link communication buffers. Comparison of these figures indicates that the bidirectional throughput was not limited by the TCP-to-TCP process, and was therefore limited by the TCP processing at either the transmitter or the receiver end. This experiment does not enable us to work out which is the constriction.
Figure 47 - TCP throughput as a function of transmit buffer size

Unidirectional throughput was a little slower, peaking at 1410 kbytes/second. This maximum was only attained for the largest application transmit buffer sizes, and tails off at a tolerable rate for packet sizes down to 200 bytes. For smaller packets, the rate falls faster, reflecting the extra overheads needed to receive small communications from the user process before combining them and transmitting 1400-byte packets over the network. The bidirectional case, above, is able to mask the bottleneck at the application interface because it keeps the link busy with traffic flowing in the opposite direction. Eventually, packet sizes drop to the point at which neither application can keep the communication channels and the output processing stages busy.

Notice that, in the unidirectional case, sending just 50 bytes from the application at a time represents a transaction rate of over 11 600 transmissions per second, justifying the inclusion of the accumulating buffers in figure 35. Without them, the main TCP ALT loop would be over-stretched.

Continuing to explore the effect of varying the TCP process parameters, the response to changing the size of the TCP transmit and receive buffers was investigated next. Data is first stored in these buffers when it is sent to the TCP process by the application; it is then moved from the send buffer in one TCP process to the receive buffer in the other TCP process when space is available in the latter. Bytes are deleted from the transmit buffer when their receipt is acknowledged by the receiving TCP process, and are deleted from the receive buffer when they are accepted into the
receive application process. These buffers, and their management, are therefore crucial to TCP operation.

Many tests have been performed whilst varying the size of the TCP buffers. In figure 48 below, the effects of different window sizes are shown. In this graph, and several other TCP graphs, five different runs have been plotted at each x-ordinate, for each trace. This is to demonstrate the variability of the results. Since the TCP process does not reset its buffer indices when a connection is closed, the next connection to open picks up essentially random values for almost all position-dependent variables. This can help us to determine whether propitious initial values, favourable buffer sizes or carefully-chosen message lengths have a beneficial effect on our measurements. By plotting the graphs with lines joining the points, to aid visualisation, the repeated runs cause some of the traces to have strange vertical sections as a consequence.

![Graph showing the effect of various window buffer sizes on the TCP throughput](image)

*Figure 48 - the effect of various window buffer sizes on the TCP throughput*

We can see here that the TCP buffer size does not have a large bearing on the throughput - on the low-latency test rig in use here, at least. Certainly, the data rate does not fall below Ethernet speeds (1187 kbytes/second or 848 TCP packets/second of 1400 bytes) while the TCP buffers are bigger than 10000 bytes.
Below this, three factors lead to a sharp reduction in performance as the buffer size is reduced:

- Firstly, the number of short TCP packets, created when user data wraps around the end of the buffer, grows because the transmit algorithm has been designed to avoid copying data to build full-size TCP frames.

- Secondly, the latency of the link causes everything to be transmitted from the transmit buffer before any returning acknowledgements signal that the buffer may be flushed and more data can be accepted from the user. This effect is also an issue in the destination TCP's receive buffer, which may run empty for periods before being replenished. Shortage of space in the transmit buffer appears to be slightly more harmful to throughput than the constraint of a small receive buffer.

- Thirdly, the TCP process implements an algorithm which only signals an enlarged receive window when it has grown by more than a specified amount since the last signal. Whilst this normally prevents the sending TCP process from transmitting unnecessary small packets, it can also have the effect of throttling communications when the overall window buffer is too small. This algorithm could easily be adjusted to accommodate TCP processes with small buffers, such as those running on T2 processors which can only address a limited memory space.

We can expect that larger transmit buffers will be needed to maintain performance as the latency of the link grows. This is referenced throughout the literature, and is seen as a particular problem - if not a reason for rejecting TCP/IP entirely - on high-bandwidth, high-latency satellite links.

Next, the system's sensitivity to the size of each transmitted TCP frame was examined. Figure 49 plots the throughput against the amount of data in each TCP segment. This time, the TCP window buffers are maintained at 30,000 bytes each, and the receiving and transmitting user processes communicate with their respective TCP processes using 1400-byte packets when possible. Since, at small packet sizes, many hundreds of packets would be generated from the 30,000-byte TCP transmit buffer, and this would overrun the overwriting buffer in the TCP-to-TCP process, the second TCP transmit algorithm was used. This only sends a few packets before re-entering the main TCP ALT loop and servicing the IP input channel, ensuring that the TCP-to-TCP buffers could cope. Using this revised algorithm explains the slightly lower maximum rate of 1372 kbytes/second, compared with the figure of 1410 kbytes/second for the more aggressive transmit algorithm.

Again, this graph shows no unpleasant trends. Throughput increases quite smoothly as the size of transmitted segments increases. The slightly raised throughput for 1400-byte packets is because the user process transmit message size and the other user processes' receive buffer communication size are also 1400 bytes, and this frequently results in a one-to-one correspondence between messages passing right along the TCP stream. There will still be a few short segments generated as
the circular transmit buffer wraps around, but these will not have a noticeable impact when so many packets are full-sized.

![Graph showing TCP throughput against transmitted segment size](image)

*Figure 49 - variation in TCP throughput against transmitted segment size*

There are three distinct phases in the graph above, although the graph in figure 50 (which represents the information in figure 49, but as the number of packets per second for differing segment sizes), makes the phases clearer.

Firstly, for packet sizes from 1 to 250 bytes, the packet rate falls linearly as the packet size increases. This is because the processing load saturates the TCP processors, representing a load of 337 µs per iteration of the output. Lastly, the throughput rate rises linearly for segment sizes between 700 and 1200 bytes. In other runs with slightly different parameters, this linear phase extends all the way to the 1400-byte segment size. This relationship is because, with large packet sizes, the amount of per-packet processing is proportionally lower and the system becomes limited by data movement and memory accesses. The measurements of the communication harness in section 4.4.1 demonstrate that it is not communication-bound. In-between, for packets of size 250-700 bytes, performance is limited by the processing capabilities of the TCP processors - they are not scheduling their high workloads optimally and therefore communication tasks which could proceed are being delayed by other processing.
Finally, since all the pipelined router, IP and network interface processes, as well as the network medium itself, delay each TCP packet as it passes from its source to its destination, the effect of long-latency paths has also been explored. The TCP-to-TCP filter process had timers added to record the time that each packet was received from its source TCP process, and a timer wait was added to stall each packet immediately before it was re-transmitted to the destination TCP process. This modification involved the addition of just four lines of occam - which worked first time - compared with considerably more grief if this task were to be implemented using most other programming languages.

Figure 51 shows how the throughput between two TCP processes degrades as delays are added to the bidirectional paths between them. The TCP processes in question used using transmit and receive window buffers of 30,000 bytes, segment sizes of 1400 bytes, and 1400-byte user application buffers. Therefore, the first point on this graph is directly comparable with the other graphs, above, which have no programmed delay. Plots are provided for both of the TCP transmit algorithms discussed earlier in this section, for unidirectional transfers only. Notice that most actual local area network delays will be much smaller than the x-axis implies - 25,000 $\mu$s corresponds to a round-trip distance of over 7500 km at optical fibre speeds. Chapter 5 shows that an Iserver connection running across an Ethernet is capable of 285 host server interactions per second (or 3500 $\mu$s per interaction), and this includes two network transmissions, all the protocol
processing, plus scheduling the Iserver process in user-space on the host workstation. Even for network latencies of 1750 µs per one-way transmission, this delay would still allow two of my TCP processes to communicate at around 1400 kbytes/second. Thus, local area network latency should not be seen as a problem. [Jacobson92] concurs. To support this, from another viewpoint, [Smith91] noted that the mean time which packets spent in the transmit queue of a conventional workstation exceeded 1000 µs for Ethernet loads above 15%, but did not rise above 2000 µs for any loads below the maximum 30% network load which he was able to measure.

As a conclusion, the TCP process appears to be well-behaved and well-tailored to the particular transputers being used. When configured with large buffers, when allowed to use TCP segments which are the full size of an Ethernet packet and when driven by applications which are prepared to communicate using packets of 1000 bytes or more, it is able to communicate with its peers at rates well in excess of that of Ethernet.

Next, we look at the lower processing layers to see whether they can support these communication requirements.

![Figure 51 - the effect of transmission delays on the TCP process](image-url)
4.4.3. Throughput of the TCP Routing Processes

The TCP root and tree router processes are not end processes in the TCP/IP pipeline, so they cannot be tested in the same way that the TCP process was in the last section. Instead, the TCP router processes are driven by small test routines which drive TCP-shaped and IP-shaped packets through a single instance of each router process, as shown in figure 52. The precise contents of each packet are irrelevant, except for certain IP and TCP address fields, and the checksum values. When testing the tree router process, which is not shown in figure 52, the protocols on the left-hand side must be changed to \texttt{TCP.TX} and \texttt{TCP.RX} since only the root router is responsible for converting between the IP and TCP protocols.

![Diagram of testing the TCP root router process performance](image)

\textit{Figure 52 - testing the TCP root router process performance}

Again, we shall vary the sizes of all the interesting variables and watch the effects in the measurements below.

Initially, the throughput of the root router and the tree router processes was obtained with just a single channel on the right-hand side of the router process, corresponding to no multiplexing or demultiplexing in this stage. With each router, in turn, installed in the configuration above, the generator process was used to create packets of the appropriate protocol and with varying amounts of TCP data. The graphs of results, below, have x-axes labelled with the size of the IP packets carried in each case; the IP packet carried 40 bytes of TCP and IP headers in addition to the TCP data, whilst the TCP packets only carried the TCP header of 16 bytes plus the TCP data. The plots, therefore, are for packets containing between 0 and 1400 bytes of TCP data, but the y-values shown are for the throughputs of real network traffic as intercepted at the IP process.

In the TCP root router, the Internet addresses and the TCP port addresses in each packet passing from the IP side to the TCP side are used as search keys into the routing table. These yield the address of the channel which connects to that packet's TCP destination process. Each packet's
TCP data payload may also have its checksum tested by the root router, depending on the checksumming strategy selected. Only the checksumming part of this activity might be performed in each TCP tree router process, since the routing table lookup is only performed in the root router. Groups of packets of each size (from 0 to 1400 bytes, plus the IP and TCP headers) were transferred both ways across the two types of router process, and the transmission rate was measured. Once each group of packets had been sent from the IP simulator to the packet receiver in the TCP simulator, timings were calculated and sent back to the IP packet receiver in a special packet with a tag which told the router to pass it through unchanged. Then, the TCP packet transmitter sent the same number of similarly-sized TCP packets back to the IP packet receiver. Routing decisions and checksum computations were performed in each direction. On receipt of the returning packets at the IP packet receiver process, more timings were computed and the results were recorded. Although checksums would be calculated just once as each packet passes through the TCP routing hierarchy in a real application, the checksums have been evaluated in each of the router processes installed in this experimental harness. The measurements were performed on three 20 MHz T800 transputers, using 5-cycle memory, and the results are shown in figure 53.

![Figure 53 - throughputs of the root router and the tree router processes](image)

Surprisingly, the throughputs of the root router and the tree router processes are very similar. In all cases, the throughput matches the capability of the TCP process, as shown in figure 49 above, with transmission rates of over 1400 kbytes/sec for maximum-sized TCP packets. There is a marked
reduction in throughput for smaller packets, due both to the store-and-forward buffering elements in the pipeline and to the constant parts of the routing algorithm which do not depend on the packet size.

Achieving the performance seen here was not easy. As with all the inter-transputer communication reported in this chapter, high priority double-buffers were installed on both sides of the transputer links to keep data flowing smoothly. Again in this case, removal of these buffers had a large effect - about a 20% throughput loss. More importantly, early attempts at measuring the TCP router processes returned figures of under 900 kbytes/sec, which were unacceptable. The reason was found to be a poor default memory layout - the code for the checksum process was being loaded into the transputer’s slow external memory. In these circumstances, the checksum code accesses together with the memory accesses of all the data communications saturated the memory bus of the transputer. Re-compiling the offending checksum routine, with the #PRAGMA LINKAGE option enabled, immediately gave an improvement in performance of nearly 50%.

Other router process measurements are needed to test for throughput degradation when the routers are used as multiplexers and demultiplexers, rather than being used to carry just one stream of packets as above. This time, the numbers of TCP simulator processes would ideally be varied, but there was no convenient mechanism for placing them all on separate transputers (in place of Transputer 3 in figure 52), because transputers only have four links. Instead, it was necessary to place all the TCP simulators together on Transputer 2, together with the router process under test, as shown in figure 54.

![Figure 54 - exercising the TCP routing processes using multiple streams](image-url)
Apart from the TCP simulator being tested, all the other simulators were very cut-down versions which only responded to initialisation packets, to save memory. The TCP simulator processes for the channel of the router which was being measured still resided on "Transputer 3". Thus, the following four graphs are not comparable with figure 53, due to the extra quiescent processes on "Transputer 2", but may be compared with each other to see the effect of increased numbers of multiplexed channels. Between one and twenty channels were connected to each router process, in turn.

Figure 55 - tree router performance - TCP towards IP

The main task of the tree routers is to perform checksum computations, when required (but it was always done when generating the figures for this set of graphs), and to propagate packets to and from the higher layers of the TCP tree hierarchy. As the number of connected channels increases, the main incremental expense in both the transmit and receive tree routers is the extra channel added to the ALTs of the main multiplexer and demultiplexer, pictured in figures 31 and 32. Since these single-transputer trials passed between 1000 and 2000 packets per second, executing 2000 ALT guards per second was a small cost and the throughput remained high. As the number of connected channels was increased (to 20 in these measurements), the total number of ALT guards processed per second rose to 40 000. The Transputer Databook [INMOS89a] states that execution of an ALT guard takes between approximately 20 and 33 processor cycles, depending on whether the program and/or its workspace data is stored in slow off-chip memory. Channel indexing takes even longer. This represents between 4% and 7% of the total processing capability of the
transputer, and agrees closely with the measurements in [Shea90]. It is therefore rather less
significant than the difference in packet storage costs as the data throughput varied between
80 kbytes/sec and over 1000 kbytes/sec for the different packet sizes. At the top of this range, the
total of the receive and transmit communications consumes up to 25% of the memory bus
bandwidth. In addition to both of these, the packet checksumming will account for more than 20%
of the processor resources, depending on how much of its code is stored in on-chip memory.

Looking first at the TCP towards IP direction, in figure 55, the effect of the greater ALT costs can
be seen in the nest of curves. Although it is difficult to differentiate the individual curves from their
point marks in the graph, the traces really are nested, with the one-TCP case at the top and the
twenty-TCP case at the bottom. This is true for each of the graphs in figures 55 to 58. Between
each trace is only the half-percent or less of difference that we predict above from an increment of
2000 ALT guards at 33 processor cycles each. There is also a large gap between the fifth and sixth
curves. This is due to the eviction of a critical part of the router process code - the checksum
routine - from the transputer’s on-chip memory into slower off-chip memory. Each time that
another channel is added to the router process, about 70 bytes of extra high-priority buffer code and
workspace are required. Some of the other trace spacings are more widely-spaced, too, as the extra
memory requirements cause more words of the router process to overflow into off-chip memory.

![Figure 56 - tree router performance - IP towards TCP](image)

Figure 56 - tree router performance - IP towards TCP
Figure 56 shows the tree router's performance when passing packets in the IP-to-TCP direction. Again, the incremental cost of the extra ALT guards is seen, together with the same disruptive gaps caused by the memory placements. Apart from this, no other problem is highlighted by the increasing degree of the multiplexers and demultiplexers.

![Graph showing router performance](image)

*Figure 57 - root router performance - TCP towards IP*

The root router process does a little more work than the tree routers. In addition to all the tasks of the tree router processes, the transmit side of the root router must maintain the routing table - a small extra duty. Not surprisingly, therefore, the graph in figure 57 exhibits no major differences from that in figure 55.

Finally, the receiving side of the root router process looks up the header fields of incoming packets in the routing table, in order to provide them with the address which they use to navigate through the tree routers above. Since the cost of each additional table lookup is only that of comparing two INT32 values and two INT16 values, again this will be difficult to measure when only up to 2000 packets are being passed every second. This is borne out by figure 58. The similar position and size of the upper gap between the third and fourth traces in the two root router graphs is due to code being evicted to off-chip memory which is common to received and transmitted packets - the checksum code in this case. The different gaps around the fifteenth trace are caused by the displacement of code which is not common to the two streams.
The results in this section show that the TCP routing processes, when running on processors by themselves, are capable of transmitting large IP packets at data rates of more than 1400 kbytes/second. This balances well with the capabilities of the TCP processes, in similar circumstances. In order to attain these rates, it has been necessary to pay careful attention to the placement of workspaces and process code in the transputer's on-chip memory, although the size of both router processes stops them from fitting together into the 4 kbyte on-chip memory of a T800. Rates drop, of course, when transferring smaller packets since the number of alternation constructs increase and the loop and packet overheads become more significant.

When large numbers of channels are multiplexed, the throughput falls as the number of alternation guards increases. [Shea90] explored the issue, demonstrating that trees of cascaded multiplexers were more efficient than a single multiplexer when combining more than about ten channels. A tree of four-input multiplexers was shown to be the optimum, with three-input ones following closely behind; both of these could halve the cost of a twenty-input multiplexer. Coincidentally, my tree routers are used as nodes of ternary trees when building trees of transputers, and therefore exploit this cascading technique.
4.4.4. Throughput of the IP Process

The IP process was installed in a test rig which looks very similar to that used to exercise the TCP router processes. It is shown in figure 59.

![Figure 59 - harness for the IP performance measurements](image)

The only parameter to the IP process which can be varied and which affects throughput significantly is the packet length. As with the TCP root router, the outgoing routing table look-up (this time, dealing with IP to Ethernet address translation) will become more costly as the number of entries increases, but the extra increments will be small and buried in the noise of the measurements.

The effect of packet size on the throughput to and from the network is shown in figure 60. As with the TCP router graphs, the x-axis in this graph shows the Ethernet packet size; the IP process strips the 14-byte Ethernet header from each packet before passing it upwards to the higher-level processes, and adds 14 bytes in the opposite direction. The graph shows that the limited IP processing requirements place little load on even a slow 20 MHz transputer. The curves (for both incoming and outgoing traffic) remain flat and communication is link-bound for all network packet sizes of 400 bytes or more, which are thereby able to transport more than 1.6 Mbytes/second of network traffic. As the packet size declines, the number of packets per second increases, and the packet header processing time begins to dominate. This is inevitable, and should cause no problems since large packets are exchanged by processes designed for high throughput.

The IP process does not appear to be a bottleneck, and could be a prime candidate for running on a single transputer with the root router process, maybe provided that the latter offloads some of its checksum processing to a tree router process. It might also be possible to run the IP process on the transputer which controls the Ethernet hardware, although if this is a 16-bit transputer, the memory requirement of the IP process could be too much of an imposition.
4.4.5. Throughput of the Whole TCP/IP Protocol Stack

Having investigated the performance of each element of the TCP/IP protocol stack individually, now is the time to bring them all together and to exercise them as a whole. Figure 61 shows the test configurations used. All measurements in this section were performed on 20 MHz T800 processors with 250 ns (5-cycle) external memory. The figures could all be expected to rise with the provision of faster processors, faster memory, or both. Throughout this section, the application processes driving the TCP/IP stacks used transmit and receive window buffers of 30000 bytes, segment sizes of 1400 bytes, and 1400-byte user buffers throughout.

A discussion of why these configurations were chosen, and how they performed, follows:

In Stack 1, one copy of each of the application, TCP, root router and IP processes were compiled together, and run on a single transputer. This transputer was connected to another running an identical stack of processes, with an instance of the IP-to-IP buffer (see section 4.4.1) running in the middle on a third transputer. The application processes could be configured to generate unidirectional or bidirectional communication streams. Double-buffer processes were used on all the link interfaces between the transputers, but no buffering was used between the protocol processes on the same processor.
This configuration forms the base-case for the other measurements in this section of the thesis. The application-to-application throughput was 583 kbytes/second for unidirectional traffic, which rose to a total of 650 kbytes/second for bidirectional traffic. These figures are presented in table 3. The reduction in throughput from the order of 1400 kbytes/second of the TCP-to-TCP measurements in section 4.4.2 was due to the additional execution of the TCP router and the IP processes together with the TCP and application code on a single transputer, allied with the increased competition for on-chip memory.

The first division of processing effort was explored by moving the IP processing onto a new processor, in Stack 2. However, this only unloaded a little of the processing, and replaced it with the management and extra memory usage of a set of high-priority double buffers on the transputers running the TCP processes.

By doubling the number of transputers, performance was increased by about 20%, to 700 kbytes/sec for unidirectional traffic and 25% (to 797 kbytes/sec) for bidirectional data. The speed was not doubled, to match the increased number of processors, for several reasons. Firstly, the load balancing was clearly not even, since the IP process was the least-demanding of the three when measured in section 4.4.4 above. Secondly, the extra high-priority buffering processes were installed by the D7205 occam configurer into on-chip memory, leaving the TCP and TCP
checksumming processes with less room for their process workspaces than before. This caused the main checksumming code to reside in off-chip memory.

<table>
<thead>
<tr>
<th>Stack number</th>
<th>Unidirectional Throughput (kbytes/second)</th>
<th>Bidirectional Throughput (kbytes/second)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>583</td>
<td>650</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>700</td>
<td>797</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>716</td>
<td>810</td>
<td>both checksums in TR</td>
</tr>
<tr>
<td>3</td>
<td>817</td>
<td>844</td>
<td>both checksums in RR</td>
</tr>
<tr>
<td>3</td>
<td>724</td>
<td>867</td>
<td>incoming checksum in TR, outgoing in RR</td>
</tr>
<tr>
<td>3</td>
<td>826</td>
<td>959</td>
<td>incoming checksum in RR, outgoing in TR</td>
</tr>
<tr>
<td>4</td>
<td>1181</td>
<td>1207</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1169</td>
<td>1163</td>
<td>both checksums in TR</td>
</tr>
<tr>
<td>5</td>
<td>1156</td>
<td>1117</td>
<td>both checksums in RR</td>
</tr>
<tr>
<td>5</td>
<td>1181</td>
<td>1171</td>
<td>incoming checksum in TR, outgoing in RR</td>
</tr>
<tr>
<td>5</td>
<td>1166</td>
<td>1223</td>
<td>incoming checksum in RR, outgoing in TR</td>
</tr>
</tbody>
</table>

Table 3 - application-to-application throughput using multiple transputers

A different load-balance using two processors was obtained in the third case, Stack 3. The checksum-processing workload of the root router was sub-divided by adding a tree router, and by performing some of the checksumming in the root router and the remainder in the tree router. One of these router processes could therefore be placed on each of the transputers. Each of the four combinations of locations in which incoming and outgoing packets could be checksummed was measured, with the links between the two transputers double-buffered, as usual.

The results of this exercise were largely as expected. Firstly, we discuss unidirectional traffic from one TCP process to the other, with only acknowledgement packets flowing in the opposite direction. See figure 62 for details of this important scenario.

With both the transmit and receive checksumming performed in the tree router, and the root router used for just the TCP routing table search, only a little load was removed from the TCP processor compared with the corresponding experiment using stack 2. This yielded a throughput 16 kbytes/sec faster than for stack 2, at 716 rather than 700 kbytes/sec.
Migrating the transmit checksum to the root router process meant that the checksum in received packets continued to be calculated by the tree router process, in contention with the TCP process, so unidirectional performance was little changed (at 724 kbytes/sec). In this unidirectional benchmark, transmit checksumming was only performed on acknowledgement packets.

Moving the receive checksum calculation onto the same processor as the IP process eased the contention and allowed traffic to run at 826 kbytes/sec, which was a worthwhile improvement. Running both checksums in the root router process, rather than just the receive checksum, caused throughput to fall to 817 kbytes/sec as the processing of acknowledgements again joined the main checksumming activity.

Looking at figure 62, the four positions of the data packet checksum processing can be seen, together with the stream throughputs. Bear in mind the direction of travel of the TCP data, and that the whole of the pictured pipeline runs at the speed of its slowest element. The main correlation between those measurements with slower and faster data rates occurs when the checksum processing in the receiver side is performed in the ringed locations. This suggests that receive processing in the TCP process is a heavier workload than transmit processing, which is in agreement with [Clark89]. It derives from the need for TCP to examine each received packet independently of any other, whilst a stream of transmitted packets will have most fields in their headers constructed just once, outside of the inner packet transmission loop in the TCP process.

By moving the receive checksum processing from the processor hosting the TCP process to the one running the IP process, the receive bottleneck is exposed and reduced. Notice that no code positioning or memory layouts are changed by migrating the checksum operation, since this is performed by changing two Boolean variables in the routing algorithms and the code remains constant. Also notice that there is a minor correlation between the measured throughputs and the

---

Figure 62 - unidirectional traffic across stack 3, showing checksum locations
location of the acknowledgement checksumming, which accounts for the difference of 8 or 9 kbytes/sec within each pair of ringed entries, above.

For bidirectional traffic in the configuration of stack 3, the flows between the two TCP processes should be symmetrical, and this is confirmed since the throughputs of the streams flowing in each direction run at approximately the same rate, typically within a margin of one percent. We can therefore examine the differences in throughput as we move the checksum processing around and make direct conclusions about processing activity.

Starting with both checksums being calculated in the tree router processes (which were running on the same transputers as the TCP processes), a similar throughput was achieved as with stack 2 - at 810 kbytes/sec, just 13 kbytes/sec faster since the TCP route table searching was now being performed on a separate processor. This is evidence that the transputers carrying the TCP processes were fully utilised.

Moving just the transmit checksum processing into the root router process increased the throughput to 867 kbytes/sec, and shifting just the receive processing, to 959 kbytes/sec. This is consistent with the unidirectional case, above, where there was a small gain in the first case (less than 10 kbytes/sec) and a larger one in the second case (close to 100 kbytes/sec). In the corresponding bidirectional cases, the gains of 57 kbytes/sec and 149 kbytes/sec are larger since the increased activity of bidirectional traffic reduces the time when the transputers are starved of work due to communication.

Finally, computing all the checksums in the root router processes, which were running together with the IP processes, achieved only a marginal performance gain - 844 kbytes/sec compared with the original 810 kbytes/sec. Since sharing the checksum processing between the processors, above, yields higher throughputs, this is evidence that the transputers carrying the IP processes were saturating in this last case.

Stack 4 added a third transputer to each protocol stack, in the first attempt at allocating the processes to three processors. This placed the TCP, root router and IP processes on separate transputers, with double buffers surrounding each hardware link. Initially, in stack 4, all the checksum processing was performed in the single root router process, on just one processor.

The gains here were gratifying - over 1 Mbyte per second was transferred unidirectionally and bidirectionally, for the first time, as can be seen in table 3. The difference between stack 2 and stack 4 is that the TCP and router checksumming duties were split in this latter case, and the effect was beneficial. This led to the final benchmark where the checksumming was again sub-divided.

Stack 5 therefore uses four transputers, one for the application and the TCP process, one for a tree router, one for the root router and a fourth for the IP process. Checksum processing was split between the two router processes. This configuration was as capable as stack 4 for unidirectional
traffic, reaching 1181 kbytes/second, and only marginally better for bidirectional traffic, attaining 1223 kbytes/second. Again, the processes and throughputs for the different checksumming locations are illustrated in figure 63.

When presented in this fashion, it is possible to see that similar characteristics are exhibited by stack 5 and by stack 3. Firstly, the two elements of each set of ringed measurements differ from each other by 12 kbytes/sec and 10 kbytes/sec; this is due to splitting the acknowledgement checksum processing onto a different transputer from the data checksumming. The differences are almost exactly the same, in proportion to the data rate, as the differences in stack 3 (of around 8 kbytes/sec).

Secondly, the difference in throughput between the top ringed pair of measurements and the bottom one is about 15 kbytes/sec, which reflects the difference between performing the data checksumming in the tree router process and in the root router process. This figure is, again, closely in proportion to the throughput gain of 13 kbytes/sec achieved when migrating the root router table searching away from the checksumming activity (in the discussion of stack 3, above). Since the IP and TCP processes are placed on separate transputers in stack 5, there are no other factors to complicate the analysis of its checksum processing, and no large differences in data rates amongst any of the experiments.

The throughput figures for stacks 4 and 5 are particularly notable, since they are clustered near to 1187 kbytes/second which is the theoretical maximum TCP throughput across Ethernet. Thus, groups of three and four slow 32-bit transputers are able to generate enough traffic to saturate an Ethernet.

This section has described a progression of measurements which show that increased parallelism improves the performance of TCP/IP protocol processing. The performance gains are far from...
linear with the increased number of processors, however, due to the considerable overhead of buffering which has to be added at each new processor boundary. This buffering is not deleterious due to the number of extra instructions which are added, but because the high-priority buffer processes are configured (by the IMS D7205 compiler) to use the fast on-chip memory and this displaces the main protocol-processing code into much slower memory. A compiler which allowed the user to specify the memory placement more precisely would minimise this effect.

The measurements in this section led me to re-examine the tolerance of the TCP process to propagation delays in the intermediate protocol-processing software and network hardware, first shown in figure 51. The code in stack 5 adds traversals of the tree router, the root router and the IP processes, plus six double-buffers, to both the outgoing and incoming paths measured in section 4.4.2. We can crudely model this as eighteen buffers, which of course are of a store-and-forward nature, if we ignore the communication of any protocol tags and length indicators between processes. Just considering the memory accesses and the link transmissions implies that a 1400-byte packet will take 800 μs to cross a transputer link from the memory of one buffer to that of the next (at 1.74 Mbytes/sec), and 175 μs to be copied from the memory of one buffer to the memory of the next, for a communication along a soft channel. In total, this makes a minimum of 6.9 ms for all eighteen processes. Reading this time off figure 51 forecasts a throughput of about 1.2-1.25 Mbytes/sec, which is very consistent with the best measurements of stack 5 in this section.

The effects of latency will be lower in the measurements involving fewer transputers, because these cases involve the traversal of fewer transputer links. Even case 4, crossing two fewer links, should expect to see a latency of about 4.6 ms, which translates to a throughput forecast of nearly 1.3 Mbytes/sec against a measured rate of only 1.18 Mbytes/sec. This situation, and all of those using even fewer transputers, continue to be memory- and CPU-bound.

These estimates suggest that work should be done to reduce the sensitivity of the TCP process to network latency, by improving its use of the TCP window buffers. Small further gains could also be obtained by separating the multiplexer and demultiplexer processes in the root router and running them on separate processors. This would avoid the need to use a cascaded tree router just to provide a second router process in which to perform half of the checksum processing, but at the cost of an additional link between the separated processes, which would reduce any fan-out at this stage to just two channels.

4.4.6. TCP/IP Throughput Across Ethernet

Before running real loads across Ethernet, it just remains for us to examine the performance of the Ethernet interface itself. It is simple to measure the throughput whilst transmitting onto Ethernet, using a single transputer in addition to an Ethernet module - see figure 64.
This yields a maximum throughput of 1096 kbytes/second when transmitting from a 20 MHz T800 transputer to a 20 MHz T222 transputer on the Ethernet module, due to the effect which DMA has on the communication links, as discussed in section 4.1.1.2. For the link transfer only, without the Ethernet transmission, a simulated transmission rate of 1157 kbytes/second was reported in table 1. Table 2 shows that the simulated maximum link transfer rate for the T225 processor is almost identical to that of the T222, and this has been verified in these Ethernet experiments, too.
Figure 65 shows how the transmission rate drops for smaller packet sizes, when the Ethernet inter-frame gap and transputer per-packet processing costs become more significant. Also shown is the theoretical maximum packet transmission rate.

Measuring the rate at which the Ethernet hardware and software can deliver received packets to another transputer is more difficult, due to the need to generate packets at near-maximum Ethernet rates from another processor. Since the transmit and receive software path lengths are very similar, it should be possible to assume that the receive rate is similar to the transmit one given above, at least for a T225 controller with a similar link performance in both directions whilst under DMA control (see figures 1 and 2 again). A single point measurement has been taken to substantiate this.

The 20 MHz T225 on the Ethernet TRAM was set to receive all the packets passing on Ethernet, regardless of their destination addresses, and to send them to the T800 transputer, where the packet sizes and the receive rate were recorded. By transmitting a stream of TCP/IP traffic between a pair of Sun workstations, an exceptionally high network load was created. The T225 was able to transfer 1072 kbytes/second of this to a T800, corresponding to 770 packets per second with an average size of 1391 bytes per packet. This is very similar to the transmit transfer rate, and is consistent with the simulated DMA experiments. No measurements were taken of packet reception rates using a T222 processor, due to concerns about hardware damage whilst swapping processors, but they can be expected to show a dramatic reduction compared with the T225, in line with the 829 kbytes/second figure in table 1.

The shortfall between the theoretical and recorded Ethernet throughputs of the commercial B407 Ethernet TRAM is a crucial bottleneck in the following measurements of TCP/IP performance between transputers and Unix workstations. A maximum rate of 1072 kbytes/second, at the Ethernet packet level, corresponds to only 1015 kbytes/second of TCP traffic, even before the degrading effects of acknowledgement packets and the network latency are taken into account.

The final stage of the TCP/IP performance evaluation involved measurements of the transputer configurations in section 4.4.5, but when attached to a real Ethernet, rather than back-to-back through a filter process. These are pictured in figure 66.

The same, relatively slow, T800-20 transputers were used, together with a B407 Ethernet TRAM carrying a T222-20 and (later) a T225-20. At the other end of the TCP/IP connections was a Sun SparcStation20 workstation, running SunOS 4.1.3. Preliminary tests showed that less-powerful models in the SparcStation family performed almost exactly as well in these tests, probably due to having identical Ethernet hardware. For comparison, the SparcStation20 is able to send TCP data across Ethernet to a SparcStation10 at 1087 kbytes/sec, and the SparcStation10 can transmit to the SparcStation20 at 1069 kbytes/sec, using the same program as used in the transputer trials, below.

The lack of a parallel programming environment on the workstations meant that it was impossible to collect reliable and accurate figures for bidirectional transfers between them and the transputers,
so only unidirectional figures are presented here. Only one Ethernet TRAM was available, so back-to-back transputer measurements via Ethernet could not be obtained.

The throughputs of the various test runs are shown in table 4. Each run sent several million bytes from the workstation’s user process to the transputer configuration’s application process, and then transferred a similar amount in the opposite direction. Since the network and the Sun workstation were not entirely idle whilst all these measurements were being taken, the best figures obtained from a series of runs are presented for each case.

Some of the results are presented for two scenarios - using a T222 on the Ethernet TRAM and then using a T225 in its place. In these cases, the throughput when transmitting from the Sun, via a T222 transputer, to the other transputers was around 160 kbytes/sec slower than when the T225 provided the network interface. This was consistent with the interference from DMA activity on the T222, and was not present for transputer-to-Sun transmission due to the provision of link buffering in that direction. All other measurements have been made solely with a T225, since the T222 bottleneck was both unacceptably slow and was masking other behaviours.

Another issue which arose when communicating across Ethernet was the presence of the overwriting buffer in the LANCE process on the front-end T225 transputer. When two stacks of
TCP/IP processes were connected back-to-back via IP-to-IP processes (see section 4.4.5), deadlock was avoided by incorporating large FIFO buffers and only dropping packets when these buffers filled up. In single-stream performance trials, the buffers were created large enough to ensure that packets were not lost, and throughput was not affected by TCP retries. Conversely, when using Ethernet, this buffer sometimes over-ran prematurely when the link from the T225 to the other transputers could not cope with the highest levels of incoming Ethernet traffic. It became clear that any load imbalance between processes in the TCP stack would prevent the LANCE receiver from delivering each packet to the IP process before overwriting it with the next, resulting in undesirable TCP retries and a huge drop in performance. Insertion of a double-buffered FIFO process between the LANCE process and the IP receive process cured the problem until the link bottleneck became the limit to throughput. Such FIFO buffers were used in all the experiments whose results are reported in table 4.

Further difficulties arose when running the TCP/IP processes on several transputers against a real network, as a result of inadequate load balancing. Without the extra parallelism inherent in the back-to-back process stacks of the earlier measurements, the pipeline processes in these unidirectional benchmarks were occasionally becoming idle as data packets progressed in one direction and acknowledgements returned in the other. To solve this, it became necessary to modify the double-buffers, which allowed each of the transputer links to run in parallel with the protocol processing. These were converted into FIFO processes so that more slack time could be absorbed on either side of the links. Whilst the extra buffering made the throughput more consistent, it introduced more high-priority code to each processor, and evicted more of the protocol processes into the slower off-chip memory on the affected transputers. This problem was worst around the root router process, which converts packets of protocol RX.PROTO into protocol TCP.RX and from TCP.TX to TX.PROTO, and therefore had to be placed on a transputer together with four different buffer or FIFO processes, one for each channel protocol. Occam2.5 would ameliorate the problem slightly through channel retyping - the RX.PROTO and TX.PROTO protocols are identical (except for their names) and the TCP.RX and TCP.TX protocols are also similar enough to be handled together in the same process using the new language facilities.

At this stage, it became necessary to hand-tune the memory layout of each experiment by judicious use of the #SECTION directive to the linker, to reduce the effects of the high-priority process placements. In particular, common routines such as the TCP and IP checksumming code and the procedures which reverse the bytes of words into network order were always loaded into on-chip memory. All the results reported in this section made use of non-default memory layouts, which tended to be different for each process configuration and had to be tuned manually. This has been an inevitable consequence of extracting maximum performance from pipeline configurations. It should cease to be a serious issue on the T9000 transputer and on the forthcoming T450 transputer, both of which have 16 kbytes of on-chip memory - sufficient for all the crucial process code and local workspace variables.
### Unidirectional Throughput

<table>
<thead>
<tr>
<th>Stack number</th>
<th>Sun to T800 (via T222) (kbytes/sec)</th>
<th>T800 to Sun (via T222) (kbytes/sec)</th>
<th>Sun to T800 (via T225) (kbytes/sec)</th>
<th>T800 to Sun (via T225) (kbytes/sec)</th>
<th>Notes (checksum location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>421</td>
<td>776</td>
<td>576</td>
<td>754</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>585</td>
<td>854</td>
<td>750</td>
<td>874</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>761</td>
<td>721</td>
<td>all in RR</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>763</td>
<td>850</td>
<td>all in TR</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>767</td>
<td>862</td>
<td>transmit in TR, receive in RR</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>773</td>
<td>726</td>
<td>transmit in RR, receive in TR</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>870</td>
<td>860</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>784</td>
<td>784</td>
<td>all in RR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>960</td>
<td>863</td>
<td>all in TR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>795</td>
<td>942</td>
<td>transmit in TR, receive in RR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>955</td>
<td>793</td>
<td>transmit in RR, receive in TR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>854</td>
<td>850</td>
<td>all in RR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>248</td>
<td>860</td>
<td>all in TR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>863</td>
<td>863</td>
<td>transmit in TR, receive in RR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>247</td>
<td>856</td>
<td>transmit in RR, receive in TR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default memory layouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>784</td>
</tr>
<tr>
<td>960</td>
</tr>
<tr>
<td>795</td>
</tr>
<tr>
<td>955</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory layouts optimised</th>
</tr>
</thead>
<tbody>
<tr>
<td>854</td>
</tr>
<tr>
<td>248</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>247</td>
</tr>
</tbody>
</table>

**Table 4 - TCP/IP performance across Ethernet to a SparcStation20**

Now we can discuss the performance figures themselves; having dismissed the T222 columns above, only the T225 ones are dealt with here. In **Stack 1**, which implements all the protocol processing on a single T800, we can see that the Sun-to-transputer throughput is some 180 kbytes/second slower than transfers in the opposite direction. This is due to the slower performance of the TCP process in receive mode, as noted when discussing stack 3 in the back-to-back trials, above. The stack 1 throughput in the slower direction (576 kbytes/second) is very similar to the 583 kbytes/second unidirectional throughput of stack 1 when two copies were run back-to-back, since the synchronised transfer in the latter case slows it down to that of the slowest pipelined process.

Taking the **Stack 1** figures as a baseline, **Stack 2** improved on the transfer rate by 180 kbytes/second in the to-transputer direction and 120 kbytes/second in the opposite one, by moving the IP process onto a second transputer, thereby off-loading the IP processing and
increasing the buffering in the pipeline. This effect is broadly similar to that in the back-to-back experiment, allowing for the differing latency of the configurations.

Stack 3 split the TCP routing activity across the two processors, placing the root side with the IP process and the tree side with the TCP process; this hardly improved on the performance of stack 2. Indeed, with the checksum processing in some locations, it was slightly worse. The main reason for this is that placing the root router process together with the IP process required four different types of FIFO buffers to be included, since the root router process converts between the RX.PROTO and TCP.RX protocols in one pipeline direction and between the TX.PROTO and TCP.TX protocols in the other. Very little fast on-chip memory was left for the root router or the IP processes, which slowed down accordingly.

There was very little variation in throughput as Stack 3 received traffic from a process running on the Sun workstation, regardless of where its checksum processing was done. Because of the poor memory layout for the root router processor, no gain was obtained from off-loading the checksum processing to there from the process running the TCP receiver process. When the TCP process was transmitting to the Sun workstation, the lower cost of TCP transmit processing yielded over 100 kbytes/second greater throughput when the checksum processing of transmitted data packets was performed in that processor rather than with the IP process. In both directions, an improvement in throughput of 4 to 12 kbytes/sec was obtained when the TCP acknowledgements were not checksummed in the same transputer as the data packets. This is consistent with the back-to-back measurements, above.

Stack 4 provided a better optimisation, this time using three T800 transputers together with the Ethernet TRAM. By running all the TCP routing activity in one root router process, and providing it with a processor all to itself, interference with the TCP process and the IP process was removed. The IP process was able to run together with FIFO buffers without a serious memory shortage, as in stack 2, and the TCP processing could also proceed without hindrance. Thus, throughputs of 870 kbytes/second in the Sun-to-transputer and 860 kbytes/second in the transputer-to-Sun directions were consistently achieved, with less perceived sensitivity to memory layout tuning.

Stack 5 placed the TCP process, the root and tree router processes and the IP process on separate transputers, achieving the greatest degree of parallelism in these measurements at the expense of a higher latency in the longer pipeline, and of greater difficulty in optimising the memory layout. At these high speeds, the effect of the overwriting buffer in the LANCE controller hardware often caused received packets to be dropped, slowing the throughput as SunOS started to re-send packets after rather long retry intervals. There was also evidence that the precise arrival time of packets became important; since the transputer scheduled runnable processes in the order of incoming events, this could affect the delay before acknowledgements were generated.

Firstly, let us examine the arrival rate of streams of traffic received by the transputers in Stack 5. When the tree router process was used to compute the checksum, arrival rates of 960 and
955 kbytes/sec were seen; the lower of these two was seen when the checksum of outgoing acknowledgement packets was also calculated in the tree router process. When the root router process computed the checksum, the arrival rate dropped to 795 or 784 kbytes/sec, again depending on whether the acknowledgement checksumming was performed on the same processor. These figures are entirely consistent with those reported above, highlighting the benefits of separating the root router table look-up activities from the receive checksumming (in the tree router) and of performing the data and acknowledgement checksumming in different processors. Notice how near the figure of 960 kbytes/sec is to the maximum measured transmit rate of the B407 Ethernet TRAM; 1015 kbytes/sec (of TCP data) was reported in section 4.4.6 as being the maximum that a B407 could receive, even before the effects of the acknowledgement packets passing through its T225 processor and onto the Ethernet were taken into account.

For transmit processing, all the results are equally convincing, except one. When the checksumming of data packets was performed in the root router, transmit rates onto the network of 793 and 784 kbytes/sec were achieved, depending on whether the acknowledgement packets were also checksummed in the root router process. These rates are some 60 kbytes/sec better than those for stack 3, which also separated the root and tree router processes across two transputers, but are less good (by a similar amount) than those of stack 4, which placed all the checksumming on the same transputer, away from the TCP and IP processing, where it needed a reduced amount of buffering and had a slightly better memory layout.

Again, the best throughput for transmission from the transputers to the Sun was obtained when the data checksumming was performed in the tree router process. Rather than figures with a difference of around 10 kbytes/sec depending on whether the acknowledgement packet checksumming was performed in the tree router or the root router, throughputs of 942 and 863 kbytes/sec were measured in some runs. In other runs, with slightly different network and workstation activity, but identical transputer code, 876 and 863 kbytes/sec were seen in place of these higher rates. The latter values are separated by 13 kbytes/sec - the benefit from moving the acknowledgement processing. 942 kbytes/sec is an anomalous figure, which I believe is caused by a particularly fortunate order of scheduling of the transmit and receive processes on the T225 LANCE controller, and is related to the order of transmission of packets on the Ethernet. Even when using slightly faster processors, below, this rate has not been equalled or bettered.

The difference of around 80 kbytes/sec between throughput rates when the data checksumming was performed on the tree and root router processes is attributable to the processing load of the root router table look-ups, as noted above, and the large amount of high-priority buffering on the root router processor. This difference may be eliminated, yielding figures of around 850 kbytes/sec for almost all the directions of communication and checksum positions in stack 5, by careful attention to memory layout and by use of the #SECTION linker directive. These improved measurements are given at the bottom of table 4. Unfortunately, the configurations which yield the two best
values in the early results for stack 5 proved incapable of extracting packets from the LANCE receiver as fast as the Sun sent them, as shown by the later results. This caused the LANCE to overwrite packets, and the TCP processes to lapse into a slow retry mode.

In addition to the complete sets of measurements, above, several trials have been performed with 22.5 MHz transputers (well, over-clocked 20 MHz ones!). Due to the effects of the overwriting buffer in the LANCE hardware, many anomalous results were obtained as the network throughput was increased, and most of these have been omitted for brevity. Notable results are presented in table 5.

<table>
<thead>
<tr>
<th>Stack number</th>
<th>Sun to T800 (via T225) (kbytes/sec)</th>
<th>T800 to Sun (via T225) (kbytes/sec)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>958</td>
<td>623</td>
<td>22.5 MHz for router only</td>
</tr>
<tr>
<td>4</td>
<td>959</td>
<td>623</td>
<td>22.5 MHz for router and TCP only</td>
</tr>
<tr>
<td>4</td>
<td>955</td>
<td>626</td>
<td>22.5 MHz for IP and router</td>
</tr>
<tr>
<td>5</td>
<td>956</td>
<td>863</td>
<td>22.5 MHz for root router only, transmit in TR, receive in RR,</td>
</tr>
<tr>
<td>5</td>
<td>963</td>
<td>864</td>
<td>22.5 MHz for IP and root router only</td>
</tr>
<tr>
<td>5</td>
<td>260</td>
<td>850</td>
<td>all in RR</td>
</tr>
<tr>
<td>5</td>
<td>976</td>
<td>865</td>
<td>all in TR</td>
</tr>
<tr>
<td>5</td>
<td>288</td>
<td>844</td>
<td>transmit in TR, receive in RR</td>
</tr>
</tbody>
</table>

Table 5 - TCP/IP performance across Ethernet using faster transputers

Using Stack 4, it was possible to substitute 22.5 MHz parts in place of all three transputers. Unfortunately, replacing all three slower processors created a configuration in which the overwriting buffer repeatedly consumed packets when they were sent from the Suns, regardless of the size of the FIFO buffer installed on the processor together with the IP process. Upgrading just the transputer which ran the router process increased the throughput in the Sun-to-transputer direction from 870 kbytes/sec to 958 kbytes/sec, but with a consequent drop in throughput in the transputer-to-Sun direction from 860 kbytes/sec to 623 kbytes/sec. This interchange of fortunes is due to the differing scheduling patterns of processes on the router processor. Adding a second faster transputer, either for IP processing or for TCP processing, made no change to this stable situation.
The throughput of Stack 5 was improved by the substitution of one faster transputer to run the TCP root router process. In the checksum configuration which produced the best results previously, the slight extra performance in this one stage improved the transputers’ receive rate to 956 kbytes/sec and maintained a transmit rate of 863 kbytes/sec. Adding a second faster transputer to run the IP process produced figures very similar to those obtained with optimal memory layouts in table 4. The most significant improvement was in the fastest recorded Sun-to-transputer throughput of 976 kbytes/sec - very close to the theoretical limit of 1015 kbytes/sec which the T225 and the LANCE devices on the B407 TRAM can handle.

4.4.7. Summary of the TCP/IP Performance Measurements

Section 4.4 of this thesis has presented a range of figures to demonstrate the viable separation of TCP/IP processing into a number of pipelined processes which may be placed on a collection of transputers.

Each of the processes along the pipeline have been measured separately, using synthetic traffic, altering their major variables to provide a thorough indication of the limits of the design and implementation. Likewise, the test harnesses which were employed have been characterised to ensure that they did not influence the results to an excessive degree. Almost all of the protocol-processing stages perform separately with throughputs greater than Ethernet rates, satisfying the original design goal of being able to saturate Ethernet using this architecture. The exception is the commercial IMS B407 Ethernet TRAM which is flawed in not quite being capable of communicating Ethernet traffic along transputer links to other transputers at full Ethernet rates. This is due to interference between the direct memory access activity of the LANCE Ethernet controller and the transputer’s links.

After measuring the individual protocol processes, they were combined into complete stacks, utilising between one and four transputers. These stacks were run back-to-back in pairs and, individually, connected to a real Ethernet. The back-to-back performance varied by a factor of two when running on one and four transputers, attaining a one-way transmission rate of 1181 kbytes/sec between application processes running at either end of pipelines of both seven and nine 20 MHz transputers. This rate is almost exactly the maximum which can theoretically be achieved over a 10 Mbps Ethernet. Analysis of the bottlenecks in this pipeline concluded that the consequences of the occam protocol processing in the root router process, rather than the router algorithm itself, were responsible for restricting the maximum throughput.

Many measurements of the architecture’s performance in conjunction with a real Ethernet were reported in section 4.4.6. All trials exchanged data between 20 MHz transputers and an application running on a Sun SparcStation20 workstation, via a B407 Ethernet TRAM. Again, a useful increase in throughput was observed when the protocol stack processes were run on between one and four transputers. The lowest figure recorded was 576 kbytes/sec for transmission to an
application process residing together with all the TCP/IP software on just one transputer. In the opposite direction, 754 kbytes/second was achieved for the same hardware configuration. Three transputers could be used to increase the throughput to more than 860 kbytes/sec in each direction.

![Four T800 transputers](image)

*Figure 67 - four transputers yield the highest Ethernet throughputs*

Four transputers, as shown in figure 67, yielded configurations capable of transfer rates up to 960 kbytes/sec. Faster 22.5 MHz transputers improved the maximum transputer reception rate to 976 kbytes/sec, and removed some of the sensitivity of these results to the lack of buffering on the Ethernet TRAM.

The best throughput figure, above, is only 4% below the maximum rate which the IMS B407 Ethernet TRAM can transfer packets between the network and the transputers. That this difference is so small is a credit to the overall software design, since all the pipeline stages are well-suited to the hardware configurations used and to the rates obtained.
5. A Networked Transputer Facility

One of the initial motivations for this research was the perceived need for a host-independent transputer software development environment, especially to support large numbers of student users.

The original INMOS compiler products all ran on transputers, and therefore had to be run on a host machine which incorporated a plug-in transputer board (hereafter, the B004 board, after the first such card to become commercially available - the IMS B004). See figure 68 for how a PC, the B004 and a network of other transputers fit together. The compiler (or linker, or library manager) was loaded into the B004's transputer, by the host computer. The latter would then run a server which satisfied input/output requests from the transputer, passing them to and from the screen, the keyboard and the filing system. On the cheapest development system (the IBM-compatible PC), the PC itself was totally committed to running the server, and also was the performance bottleneck in that combination. Digital Equipment Corporation microVAX minicomputers could run multiple servers in parallel, but the multiple transputer cards required to support them were prohibitively expensive, and the performance was still host-limited.

Although INMOS compiler products are now available for Sun and PC hosts, the binary code which they produce still has to be run on transputers, and a host-independent way of accessing these transputers is still attractive.
The concept of replacing the host processor with a small group of networked transputers was therefore developed. This allows multiple hosts (such as Unix workstations) to connect to the transputers across a local area network. Each target transputer's I/O requests are then passed back and forth across the network to a filestore server which runs on the hosts (see figure 69). Since the incremental price of an extra target transputer (and maybe extension of the underlying tree of processors) is far lower than that of adding another PC with a commercial plug-in board, this new architecture is particularly cost-effective. Additional benefits arise from sharing the host activity between many workstations and from being able to access transputers remotely rather than having to sit at a specific PC's keyboard.

![Networked Transputer Facility Diagram](image)

*Figure 69 - a networked transputer facility*

The high latency due to the longer route between the target transputer and the host does not detract too much from the overall system performance since another big bottleneck is replaced - that between the original transputer card and its enclosing PC. [Peel90a] first introduced these points.

In order for the new networked facility to work, it must provide exactly the same behaviour on the channels which connect the host I servers with the target transputers as the B004 architecture does in the standalone system. This is easy to accomplish, since the INMOS SP protocol which runs along these channels is well-defined and is easy to support on a range of host operating systems.
5.1. The Networked Iserver Architecture

The networked Iserver is designed to run on a tree of transputers connected to an Ethernet. A single-processor version (capable of supporting four target transputers) is shown in figure 69. Greater performance may be obtained from running the TCP/IP processes on an extended tree of transputers, such as the one shown in figure 70. The leaf nodes of these trees are target transputers which may be reset, loaded and started running independently by the client Iserver processes running on the next-lower layer of transputers. Figure 70 supports a large population of target processors by running the client Iserver processes on several transputers.

File service is split across the network; the user runs one part on a host Unix machine somewhere, and the client Iserver runs on the networked facility. File and I/O facilities are provided for the target transputer by the client Iserver, in the same way that the file server on a PC would to a conventional transputer card, but the actual filestore accesses are performed back on the host processor. Connection between the host and client Iserver processes is made by two TCP/IP streams, whose purpose is discussed below.

The host Iserver is a cut-down version of the INMOS PC-based offering, and contains just the filer and the screen output facilities. Keyboard access is provided rather differently, via a buffer at the client end of the connection, to avoid polling for keyboard activity across the network. Managing different host I/O conventions has proved tiresome - the networked Iserver provides several command-line options which help it to deal with differing filename formats (filename length, case sensitivity and so on), as well as carriage-return and line-feed generation from the keyboard.

Figure 70 - A larger networked Iserver configuration
One of the most significant differences between my networked Iserver and the standalone INMOS product is that the user must specify the action to be taken if a target program fails, as part of the main loading sequence. This strategy ensures that the target transputer that failed is re-used for the diagnosis, avoiding the possibility that another user could connect to this processor across the network in the meantime. Two boot file names are supplied to the host Iserver in the command line, one as a -sb or -sc parameter and the other in the new -sd field. These filenames are passed to the client Iserver as soon as the TCP connection is opened between them. The client Iserver immediately opens the first file, reads it and sends its contents to the target transputer, thereby booting it. Once the program is running, the client Iserver then satisfies all the application program's requests using the host Iserver to manage the filestore and standard I/O channels. When the target program terminates, the client server simply closes its TCP connections with the host Iserver, and the latter terminates, returning the Unix user back to the command prompt. The client Iserver cycles around to await another connection from the network.

Should the target program fail or the user break in, the second filename that was specified initially is opened, and its contents are used to boot the target transputer in analyse mode. At this stage, the new code is likely to be the debugger or network dump program; alternatively the Transputer Development System may be re-entered in analyse mode, ready to diagnose the error condition. Further break-ins during a user's session re-start the second transputer program, again in analyse mode. This behaviour supports the Transputer Development System (TDS3) [INMOS90], which has to be re-started in analyse mode each time after its initial invocation in reset mode.

Notice that the connection states of the controlling TCP streams are used to determine who has access rights to each target transputer. TCP/IP's reliable three-way handshake mechanism for opening connections ensures that only one host Iserver is able to connect successfully to each listening client Iserver.

5.1.1. Networked Iserver Software Structure

When the host Iserver is run, it connects two TCP streams to the client Iserver across the network, as shown in figure 71. One TCP stream is used (bidirectionally) by the host to receive Iserver requests from the client and to respond to them. All keyboard characters are sent, asynchronously, across the second TCP stream to the client Iserver which buffers them. Should the buffer overflow, characters are lost (not unreasonably) if they come from the keyboard, or are blocked if they come from a re-directed file or from a pipe.

Break-in characters are passed across the second stream as Urgent TCP data, causing them to overtake ordinary characters already held in the keyboard buffer. The buffer then sends a message to the client Iserver process which is currently awaiting input, causing it to terminate. This mechanism operates in the same way that the Unix 'telnet' program handles break-ins. Following a termination request, the server processes and the TCP connections are closed down using the
technique described in [Welch89b]. Welch's scheme has been extended to take account of the channels joining the Iserver to the host being connected by TCP/IP streams rather than occam channels, since the former may be terminated by the remote host whilst the latter should be reliable. Also, the channels to the target transputer are hard links, which can become deadlocked by a failed application, and may require resetting before re-use.

---

**Figure 71 - the networked Iserver process structure - whilst servicing requests**

---

**Figure 72 - a networked transputer facility (full process layout)**
As well as supporting a number of Iserver processes, which users may access using TCP/IP connections from across the network, the design provides support for other networked applications. Two of these - a telnet status process for the networked Iserver itself and a performance benchmarking process - are discussed in sections 5.1.2 and 5.1.3, below. At various times, other services such as a teletext server [Edwards93] have also been attached to the Ethernet via this interface. The software architecture which supports this flexibility is shown in figure 72.

5.1.2. A Telnet Supervisory Service

Since the networked Iserver software is likely to be run on transputers where all the links are interconnected to the target or protocol processors (in figure 69, for instance), there is no simple place to add connections for diagnostic purposes. The INMOS B300 product uses a memory-mapped interface on one of its transputers to drive a UART, which can then be connected to a VDU status display. Instead of this complicated solution, my networked Iserver sends streams of human-readable diagnostic information from each of the client Iserver processes (and others, such as the performance benchmark process described in section 5.1.3, and even the status process itself) to a central status process which runs in parallel with all the other client processes. Currently, this process resides on the processor which hosts the three or four client Iserver processes and the two performance monitoring processes. This diagnostic information may be individually tailored (by parameters or re-compilation) to provide the information required to trace connection states, server responses and throughputs. It is then accessed by users from across the network, using the Telnet protocol [RFC854 - Postel83], running on top of the usual TCP/IP structure.

Using telnet as the access medium makes this service more widely available than with other arrangements, such as a serial port and VDU. The only snag is that, following a fatal error in the software, no tracing information remains on the VDU's display to assist in tracking down the fault.

All the status streams from separate processes are brought together at the telnet FIFO buffers, shown in figure 73, which retain the most recent 2kbytes of diagnostic text from each TCP and application process and normally overwrite old bytes as new ones arrive. A networked user may connect (via TCP) to the telnet service in just the same way that they would open a terminal session on a conventional networked host. Typing keyboard characters selects one of the buffers for display, via the control process. The buffer corresponding to this stream is then taken out of overwriting mode, and blocks the application process if necessary so that the user terminal receives a complete display of all the current information, across the network, as it is provided. The status output from the telnet process itself (which provides information on the TCP connection which it uses) must retain an overwriting buffer, to avoid a deadlock cycle. In addition to streams of text from each client Iserver process which keep track of their TCP connections and booting status, a channel connects each Iserver to a small status process which simply scrolls the username and hostname of each active connection up another display.
In a large configuration, with several transputers each supporting three client Iserver processes, it is possible to pass all the per-session status information back to a single telnet status process that runs on the transputer which hosts the TCP root router process. High-volume status streams for monitoring individual TCP connections can continue to be provided by separate telnet display processes which reside on the transputers which run the client Iserver processes.

One deficiency with a stand-alone networked service such as my Iserver is that it is difficult to store usage information. The task could be done in three different ways, but the simple choice introduces security problems:

- As each user's host Iserver makes a connection to the client Iserver, it could write a line of logging information to a central file. This scheme would only be adequate if all such hosts were connected to the same central filestore, and if they could run their Iserver programs with sufficient privilege compiled-in to allow them access to the logging file without it having to be given public write access permissions. The scheme would not be appropriate to those PC operating systems which do not provide secure user environments.

- The transputers themselves could run the Network File System (NFS) protocols, authenticate themselves as the owner of the logging file, and write to it. This is a clean solution, which can operate in a truly distributed way. Its drawback is just the complexity of developing another set of protocol software for this relatively unimportant task.
A logging server (not unlike the Unix 'syslog' process) could be run on a computer somewhere on the network. The client Iserver could connect to this, using TCP/IP and a well-known server port number, and send a logging message. These messages could be authenticated by any number of public-key encryption techniques. This scheme does not require any extra software, but may require an extra TCP process on the transputer, unless the logging is performed sequentially at the beginning or end of the client service when the existing TCP processes are idle.

Currently, no special need has been found for Iserver logging, and the provision of such a facility has been avoided.

5.1.3. A TCP/IP Benchmark Service

Since it is convenient to be able to exercise the TCP/IP software with large data transfers, perhaps whilst other services are already running, two copies of a benchmark process usually run when the rest of my server processes are active. These benchmark processes wait for a connection from the network, negotiate the number and size of packets to transfer in each direction, and then transfer the requested data, first from the requesting host to the networked server, and then back again. Performance information is accessible via the telnet status service.

5.1.4. Performance of the Iserver SP Protocol

The Iserver SP protocol [INMOS90, and other software documentation] suffers from poor performance, due to an unimaginative design. The protocol operates in client-server fashion, with each request being sent from the transputer to the host, and each response being returned after the action is complete. This guarantees high latency, since each round trip includes traversal of a transputer link, awakening of the host Iserver, interpretation and execution of the command itself, generation of the response packet and its return across the link. To compound the problem, the maximum length of an Iserver command packet is 512 bytes, which means that only 507 bytes of application data may be transferred in either direction in a file read or write operation. Each of these small transfers suffers the latency mentioned above. Since the SP protocol is a standard, however, little can be done to improve matters, although more-recently released versions of the server have increased the maximum packet size to 1024 bytes. The considerably increased latency of the networked Iserver compounds these problems.

Some of the difficulties may be avoided by performing other host services in parallel with the application running on the target transputer, or indeed in parallel with the acknowledgement of each previous request. The former is difficult to accommodate without speculatively pre-reading file blocks. The latter modifies the semantics of the client-server model, but has the distinct advantage of hiding the latency. This is not permitted in my networked server except when a
program is being booted, in order to avoid compatibility problems with Iservers running in other environments.

My networked server avoids all polling of the user’s keyboard across the network by providing a keyboard buffer at the client Iserver. Since filestore transfers cannot be avoided in the same way, experiments have been performed to examine the effect of caching each of the files opened by the Iserver. In particular, various cache updating schemes have been explored, including pre-reading, reading larger blocks than requested and buffering write commands, with particular attention to the long latencies of the networked file access. Many of these techniques are also relevant to caching Iserver filestores on PC hosts, since their transfer performance without fast processors and DMA link interfaces is often worse than the networked Iserver described here. Just increasing the maximum Iserver packet size has an appreciable effect on performance in all environments, although it does impose server compatibility problems on user applications which attempt to benefit from it.

5.2. Networked Iserver Performance Figures

There are very few meaningful performance measurements which can be taken on the networked Iserver itself, since the SP protocol has such an impact on all server operations. The results in this section follow closely those in [Rizzo93], where similar facilities were provided by PC-hosted transputers communicating across a network using the PC’s TCP/IP software. Comparisons are also made with slow and fast standalone transputer environments running on PCs. The parameters compared are low-level packet latency, and filestore access speeds.

An introduction to the four environments is given below, together with the underlying communication performance of their network interfaces and their transputer connections, where applicable.

1) An IBM PC-AT, running MS-DOS, uses an 8 MHz 80286 processor and contains a B004 transputer card carrying a 20 Mhz T800 processor, with 5-cycle memory, connected via a 10 Mbit/sec link and an IMS C011 link adaptor to the PC’s bus.

2) A Viglen PC, using a 66 MHz 80486 DX-2 processor, runs the Linux operating system with Christoph Niemann’s ‘transputer-08’ transputer card drivers. It contains a B008 transputer card carrying a 20 Mhz T800 processor, with 5-cycle memory, connected via a 20 Mbit/sec link and an IMS C011 link adaptor to the PC’s bus.

3) Luigi Rizzo’s 25 Mhz 80386 PC, running his PCserver software, using an unspecified transputer, connected via a 20 Mbit/sec link and an IMS C011 link adaptor to the PC’s bus. The host machine on his Ethernet was a DecStation 2100.
4) My networked Iserver, running across three transputers (one T225-20 for the Ethernet interface and two T800-20s for the server and the networking software). Application programs run on a further T800-20, all of which use 5-cycle external memory. The links are 20 MHz throughout. The Unix host in these measurements is a Sun SparcStation20.

Table 6 shows the speeds of link transfers across the PC interface, and the underlying speed of the networking software, in the configurations used for these experiments. It highlights the large difference in network speeds between cases 3 and 4, and the advances in PC performance since the days of 80286 machines.

<table>
<thead>
<tr>
<th>Hardware Type</th>
<th>Link adaptor speed (kbytes/sec)</th>
<th>TCP/IP speed (kbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IBM PC-AT</td>
<td>56</td>
<td>-</td>
</tr>
<tr>
<td>2 Viglen PC</td>
<td>325</td>
<td>-</td>
</tr>
<tr>
<td>3 PCserver</td>
<td>320</td>
<td>190</td>
</tr>
<tr>
<td>4 networked Iserver</td>
<td>-</td>
<td>750</td>
</tr>
</tbody>
</table>

*Table 6 - Iserver baseline hardware performance*

Amongst the performance figures quoted in [Rizzo93] is that for loading the INMOS C compiler. This test has been re-run here, using the command `icc -1` to load the compiler and to exit immediately. The bootable compiler file is 301 kbytes in length.

<table>
<thead>
<tr>
<th>Hardware Type</th>
<th>icc -1 time (sec)</th>
<th>Transfer Speed (kbytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IBM PC-AT</td>
<td>6.9</td>
<td>44</td>
</tr>
<tr>
<td>2 Viglen PC</td>
<td>1.8</td>
<td>171</td>
</tr>
<tr>
<td>3 PCserver</td>
<td>4.0</td>
<td>75</td>
</tr>
<tr>
<td>4 networked Iserver</td>
<td>1.4</td>
<td>215</td>
</tr>
</tbody>
</table>

*Table 7 - Iserver performance - loading a large program*

Table 7 shows the rate at which large blocks of boot code were transferred from the host to the target transputer. Usually, booting does not involve the Iserver SP protocol. It is therefore a measure of the raw communication performance between the host computer and the transputer, although file access is also involved. In case 4 (my networked Iserver), the boot file is accessed using SP protocol calls, so that the `-sd` reboot-on-error facility may be implemented easily, and this extra overhead has been included, too. The lone PCs in cases 1 & 2 were limited by the speed of the PC-to-transputer link, since the link adaptors in these cases are polled; this activity consumes most of the processor's time and interferes with the disc accesses. The PCserver in case 3 was limited by the host filestore access, the network transfer speed and the link speed. Not surprisingly,
it was slower than cases 2 & 4. The networked Iserver benefited from a much higher network transfer rate, and probably a faster host filestore, but was hindered by its use of the Iserver protocol. Iserver transfers of 4 kbytes were used, the same as stated in Rizzo's paper for case 3.

The high raw bandwidth of the networked Iserver's underlying TCP/IP communication, together with its avoidance of a polled C011 link interface, made the networked Iserver the clear winner here.

Next, a program was run on all of the hardware configurations, which repeatedly called the host server's "version" request. In response, the server simply packed four bytes of operating-system-dependent information together with a status byte and returned it as a 5-byte packet. The four measurements, which reflect the latency of the whole system, are shown in table 8.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>version.bah (iterations/sec)</th>
<th>round-trip delay (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IBM PC-AT</td>
<td>3000</td>
<td>0.33</td>
</tr>
<tr>
<td>2 Viglen PC</td>
<td>7580</td>
<td>0.13</td>
</tr>
<tr>
<td>3 PCserver</td>
<td>142</td>
<td>7</td>
</tr>
<tr>
<td>4 networked Iserver</td>
<td>248</td>
<td>4.0</td>
</tr>
</tbody>
</table>

*Table 8 - Iserver performance - latency of empty packet transfers*

Here, the effect of latency is dramatic. A fast PC, with a rapid path between it and the transputer, was over thirty times as fast as my networked Iserver, and more than fifty times as fast as the PCserver. Even a slow PC was twelve times faster than the networked Iserver. This is not really surprising, and represents a worst-case comparison between the architectures. Fortunately, few transfers in the execution of most applications will be of this small size (especially if they avoid polling the host's keyboard across the network).

Iserver's SP protocol specification restricts each packet to 512 bytes, of which five are used to carry command information and the remaining 507 may be used as data. The last of Rizzo's performance figures is for writing large packets from a transputer application to /dev/null on the host. This causes little operating system activity at the host, since the data is thrown away immediately it arrives, and thus the experiment reflects the latency of SP packets which carry a full payload of data (507 bytes) towards the host, and none in return.

In table 9, communication throughput and latency are given for these SP-protocol transfers of maximum size. Again, the benefit of reducing the round-trip delay to a minimum is shown by comparing the data rates. Those of the Viglen PC are reasonably impressive - although they are still less than a quarter of the file access rate that would be obtained by a program running directly
on the 80486 processor. Surprisingly, the file access rate for a transputer hosted in the Viglen PC is 50% faster than the transputer booting rate which is reported in table 7 above. Both of the networked servers were limited by the poor round-trip times of the networks, although the access rate of my networked Iserver matched that of the slow PC, which is encouraging given these delays. For real filestore transfers from a transputer, the slow PC host could only manage 25.3 kbytes/sec, whilst my networked Iserver still wrote at 66 kbytes/sec. In the latter case, this was nearly the same as for null writes, probably due to disc caching on the SunOS-controlled host processor. Communication using 512-byte SP protocol packets had a large bearing on the results in every case.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>File writing rate (kbytes/sec)</th>
<th>Round-trip Delay (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IBM PC-AT</td>
<td>71</td>
<td>7.1</td>
</tr>
<tr>
<td>2 Viglen PC</td>
<td>256</td>
<td>2.0</td>
</tr>
<tr>
<td>3 PCserver</td>
<td>45</td>
<td>11.0</td>
</tr>
<tr>
<td>4 networked Iserver</td>
<td>78</td>
<td>6.5</td>
</tr>
</tbody>
</table>

*Table 9 - Iserver performance - latency of maximum-sized packet transfers*

As an experiment to see how much file throughput is restricted by limiting the size of SP protocol packets to 512 bytes, figure 74 plots the data rates for writes from the transputers in the PCs and in my networked Iserver to /dev/null on their respective hosts, using larger block sizes than 507 bytes in the originating application programs.

These graphs show the effect of varying the block size from 50 to 1000 bytes, and what happened to the throughput when the 507-byte threshold was passed for those servers where this caused blocks to be split. In addition, a similar exercise was done with writing much larger packets to /dev/null using the networked Iserver's ability to write up to 2000 bytes in an SP packet, and also its ability to read (from a real file, this time) up to 4000 bytes in an SP packet.

The Viglen PC trace shows the relative stability of a single-user environment, although the effects of background processing, disc and virtual memory activity are visible when running Linux. It also shows the benefits of low latency across the transputer-to-PC interface. Breaking all blocks of more than 507 bytes of data into a number of 507-byte segments and a smaller one introduced an inefficiency, which is shown by the saw-tooth effect.
Since the networked Iserver is so susceptible to the long-latency path across the network, we would expect its performance to depend more on the transaction rate than on the size of each packet transferred. This is confirmed by the broadly linear traces in the graph. Above a packet size of 500 bytes, the data rate begins to rise more slowly, and the speed of the network itself begins to play a part. The network was not completely idle when these results were collected, so some of the peaks and troughs reflect activity during the several tens of seconds needed to accumulate them. There is also evidence that some of the different file block-sizes used upset the disc access patterns of the host processor, and contributed to the stepped improvement in file reading for packet sizes around 1200 bytes.

For packets of 4000 bytes, for example, the peak file access rate measured for the networked Iserver was 253 kbytes/sec, corresponding to 63.2 packets/sec or 15.8 ms per packet. Empty packets were turned around in 4.3 ms, as shown in table 8, which is 230 round trips per second. Thus, only 2.15 ms in each 15.8 ms turnaround is directly attributable to the response, and the other 13.6 ms reflects the disc access time, the operating system delays in the host, the protocol processing time in both the host and the transputer, and the request packet transmission time itself. Transmitting 4000 bytes in 13.6 ms corresponds to a data rate of 295 kbytes/second, against a measured network rate of at least twice this, so most of the discrepancy is caused by the SparcStation host.
As we can see, the relatively high latency across the network and at the host server limits the maximum throughput obtainable from the simple request-response Iserver protocol and ensures that one such stream cannot saturate the Ethernet. This means that several networked sessions may be run concurrently with a minimal effect on each other.

5.3. Networked Iserver - Summary

The networked transputer facility is a stable environment which has been used to support several classes of students and many individual projects for over two years. All of the Ion Microbeam instrumentation described in Chapter 7 was compiled and tested exclusively using the networked facility. Particular features of note are:

- The tree-structured architecture can support an arbitrary number of target transputers, and the extra routing processors which have to be added as the tree grows can be utilised for protocol processing, thus maintaining an even load balance.
- Diagnostics and management statistics are obtainable across the network, avoiding the need for a separate terminal.
- Although hampered by the high latency of the local area network, the networked service has a performance broadly comparable to that of a PC-hosted transputer, whilst offering more convenient access.
- It is self-contained, self-booting and auto-configuring on power-up.

A discussion on how the networked Iserver could be enhanced to provide multi-transputer target facilities may be found in section 8.3.
CHAPTER 6

6. Hardware Developments

Perhaps surprisingly, commercial hardware is still not available to perform two of the tasks required on the Networked Transputer Facility. Rather early in the project, it was therefore necessary to design and build a transputer board with multiple subsystem control ports, and a matching motherboard capable of supporting Ethernet TRAMs and providing feedback during booting operations.

In addition, this section briefly describes the important features of two Ethernet interface cards, built by two of the author’s M.Sc. and M.Eng. project students, which support particular points made elsewhere in this thesis.

6.1. The DTR1 Transputer board

In the Networked Transputer Facility, the transputers running the client Iserver processes must be able to reset and reboot the target processors (to which they are directly connected using links) independently of each other. This requires three sets of subsystem ports, each comprising two separately-controlled output signals (reset and analyse), and one input (error). It was therefore necessary to build a 32-bit transputer board (see figure 75) with the necessary subsystem facilities.

The specification of this board includes:

- A socket for a 32-bit INMOS transputer, suitable for the T414, T425, T800 and T805. The extra configuration input pins on the later devices (e.g. for processor speed selection) are fully supported.
- Four fully-buffered INMOS links which are protected against electro-static discharges by a diode bridge arrangement, as are all the off-board signals connected to transputer input pins.
- UP and DOWN processor control ports and four Subsystem ports, at addresses and pinouts on the main connector compatible with those on the IMS B004 and similar processor boards. The backplane connector has a pin-out which is a superset of the lower connector in the INMOS ITEM rack, allowing DTR1 boards to be plugged into this type of enclosure together with INMOS Eurocard evaluation boards.
- 2 Mbytes of 100ns dynamic memory, using sixteen 256 K*4 devices, extensible to 8 Mbytes using 1 M*4 memories.
• Full address decoding, using programmable logic devices (EPLDs or PALs) to allow for easy modification.

• A transputer memory configuration signal which is supplied at reset time from an EPLD; this can be exchanged to provide various memory timings.

• A bus expansion connector for the addition of memory-mapped peripherals on a daughter-card. This connector supplies all the transputer control lines, the 24 lower address lines in latched form, all 32 multiplexed address / data lines, 6 programmable outputs, two spare address decode outputs and appropriate reset and error lines. All outputs from the transputer to this connector are series terminated, both to provide clean output signals and also to prevent damage to the transputer should the outputs be misused.

The logic on the DTR1 board is all CMOS, and its worst-case average power consumption with a 5 V supply is about 1.5 W. This maximum value only arises when its T800 processor is frequently accessing memory at the same time as it is using its links intensively and running floating-point computations.

Production versions of the DTR1 board have been in continual use for several years, and have been extremely reliable. The INMOS ITEM rack standard for link and control signal interconnections relies on the power supply wiring to provide an adequate ground reference to all interconnected boards. It is important to ensure that the ground connections are satisfactory, to ensure correct operation of the links.

---

*Figure 75 - DTR1 block diagram*
6.2. The DTR2 TRAM Motherboard

Transputer modules incorporating an AMD LANCE Ethernet controller may be purchased from INMOS (the IMS B431; formerly the IMS B407). No vendor supplies a suitable motherboard into which these may be plugged, and so a special-purpose board was designed (figure 76). Constructing a motherboard gave me the additional advantage of being able to accommodate a block of read-only memory to contain the network boot code described in chapter 5, in addition to eight diagnostic lights and a watchdog timer.

Since the motherboard is not intended to support multiple TRAMs, it does not provide any link reconfiguration facilities as specified in the INMOS motherboard specification [INMOS89b, chapter 5]. Its full characteristics include:

- A similar form factor to the DTR1 processor board (single-extended Eurocard), and a similar ITEM rack connector and pin-out.
- Support for a single size 8 TRAM, eight size 1 TRAMs or any combination of intermediate lengths.
- Buffering and electro-static discharge protection for all four transputer links on TRAM site 0, bringing them to the backplane for external connection.

8 EPROM sockets, whose contents (a maximum of 512 kbytes) may be transmitted serially to any of the TRAM links following reset. This facility may be used to boot the whole of an interconnected transputer network, or (as in this case) just to provide basic Ethernet transmission and TFTP booting capabilities for an Ethernet TRAM.

*Figure 76 - DTR2 block diagram*
8 light-emitting diodes (LEDs) driven from a latch attached to the boot link. These provide a low-level diagnostic capability which can provide feedback during TFTP booting when no other mechanism is available.

A watchdog timer which may be used to re-boot the transputer network if no output to the LEDs has been seen for a period of 5 seconds. This is a necessary feature of many embedded control systems, which this motherboard was also designed to support.

6.3. A T414 and AMD LANCE-based Ethernet controller

One of my original inspirations for connecting local area networks to transputers came whilst devising a project for Piers Shallow, an M.Sc. student, whom I subsequently supervised. Piers very successfully interfaced the original DTR1 board prototype (seen in section 6.1) to an AMD Am7990 LANCE Ethernet controller [Shallow87]. The block diagram of this development is shown in figure 77.

![Figure 77 - A 32-bit Ethernet interface based around an AMD LANCE](image)

Briefly, the specification of this device includes:

- The full capabilities of the attached DTR1 card - 32-bit transputer, memory, subsystem ports and programmable LED outputs, etc;
- Am7990 LANCE and Am7992 serial interface devices [AMD86], which provide Ethernet connectivity via a 16-bit memory and I/O bus;
- 32K * 32 of fast SRAM memory, accessible by the transputer (directly) and by the LANCE (under DMA control);

This board - a 32-bit design - pre-dates the 16-bit IMS B407 and IMS B431 TRAMs from INMOS. There are both advantages and disadvantages to a 32-bit design, some of which have only become
apparent to me as a result of recent specification work for a new Ethernet product, although many were clear to Piers and myself while the original project was running.

- Connecting a 16-bit peripheral device to a 32-bit transputer is not particularly difficult if the transputer can always be programmed to use just the bottom 16 data bus lines. Unfortunately, in order to use block move instructions, this is not possible in this design, and 32-bit memories must be supplied on the LANCE board.

- The LANCE uses direct memory access (DMA) to read and write Ethernet packets, as a bus master device. Together with the pathways required for the memories above, this leads to the need for ten 8-bit bidirectional buffers to connect all the devices in all necessary permutations. This is a huge overhead.

- The availability of 128 kbytes of memory accessible to the LANCE allows it to be operated with two 32 kbyte buffer rings - one for transmitted packets and one for received packets. This is not possible on the B431, which has less memory and has to use some for program storage. In addition, the 2 Mbytes of memory on the DTR1 board is ample for all the TCP/IP and application code, so complete embedded applications which do not require multi-transputer performance may be constructed around a single transputer.

- The 32-bit data-paths on the transputer substantially improve the TCP/IP processing and packet copying operations, compared with a 16-bit transputer. Many of the DMA issues discussed in section 4.1.1.2 would be far less serious with a 32-bit transputer, due to its greater instruction buffering and less-frequent link-engine memory accesses.

The overall complications of interfacing the 16-bit LANCE to a 32-bit transputer make this design unattractive now that better alternatives exist. Sadly, many of the inconvenient features of the LANCE have been propagated forward to AMD's later 32-bit controllers, although these should be far easier to interface to a 32-bit transputer.

6.4. A T222 and Fujitsu NICE-based Ethernet controller

As discussed in section 4.1.1, LANCE-based Ethernet controllers have the disadvantage of an intrusive direct memory access scheme. Also, each byte transmitted or received provokes two memory operations whilst travelling between a secondary transputer and the Ethernet. Kjetil Hamre undertook an M.Eng. project in 1994, under my supervision, to identify a better solution.

Kjetil and I eventually settled on the Fujitsu MB86960 (NICE) device [Fujitsu93] as a viable alternative to the LANCE. We then built a prototype circuit, based around a T225 processor, which provided a basis for my software drivers in section 4.1.1.3. The block diagram of Kjetil's hardware is shown in figure 78 and in greater detail in [Hamre94].
The NICE Ethernet controller chip is designed for interfacing to an IBM PC. To this end, it does not use large areas of memory-mapped address space, as the LANCE does, but instead a small number of I/O ports. Most of these I/O ports provide access to 16-bit status and control registers. One register - the data register - is a 16-bit window onto the receive and transmit FIFO buffers which are maintained by the NICE hardware in a private memory which is not otherwise accessible to the controlling processor. Sequential accesses to the data register read or write to one end of these FIFOs, whilst the Ethernet controller writes or reads from the other end.

The outline specification of Kjetil's hardware is:

- A T225 transputer has access to the NICE chip and 50kbytes of memory. It also controls a subsystem port and a bank of diagnostic LEDs.
- 64 kbytes of fast private static RAM is attached directly to the NICE controller. This memory is accessed by the transputer and by the Ethernet electronics through an on-chip arbitrator. The transputer, using 3-cycle memory timing, does not have to insert additional wait states when accessing this memory, due to its speed and buffering.
- The data register on the NICE device is accessed when any one of 1024 consecutive words of the T225's memory are read or written. This clever memory-mapping arrangement allows repeated accesses to the data register to be used as the source of transputer output instructions or as the destination of inputs. Another transputer may therefore communicate directly along a link to the Ethernet FIFOs, at the cost of a single memory access for each 16-bit word on the network.

The simplicity of the NICE controller makes it a good choice for use with a 16-bit transputer, especially with the memory-mapping scheme described above for data register access. Sadly, it is difficult to perform meaningful TCP/IP communication using the limited memory and processing
power of a 16-bit transputer, which implies a two-transputer solution is still needed for most embedded applications.

Since connection of the 16-bit NICE to a 32-bit transputer would be futile (for the same reasons as for the LANCE, above), there are considerable advantages to incorporating the NICE chip with the forthcoming SGS-Thomson T450 processor. The T450 will offer a configurable memory bus, enabling 32-bit bulk memory to be used to accommodate program and workspace areas, whilst also providing memory segments which can be interfaced to 8 and 16-bit peripherals. This is likely to lead to an effective single-processor, high-performance platform for networked embedded systems.
CHAPTER 7

7. A Networked System using Embedded Transputers

In order to create a networked embedded instrumentation system, I have collaborated with the Solid State Devices and Ion Beam Technology research group in the Department. The exercise involves the control of a scanning ion microbeam, capturing signals from Rutherford Backscattering and particle-induced X-ray emission sensors and preprocessing of the received signals into images and spectra. This is done on a pair of transputers. Transmission of these results to a Unix workstation is performed under control of the transputers, which run the TCP/IP software described in Chapter 4, with support from a third transputer which provides an Ethernet interface. Finally, the results are displayed in graphical form under X-windows. The transputers and experimental equipment are controlled from a Unix workstation, across the network, and the transputers may be booted from the network, allowing rapid software updates during development.

In the version of the work reported in [Millen93] and [Peel94], I was responsible for the overall system design, the hardware and software architectures, all the code on the transputers - that which performed the signal preprocessing and all the TCP/IP network communication, as well as the 'C'-language network interface code for the display routines. David Millen designed and built the scanning hardware and wrote the graphical user interface code. The hardware and software structure used is illustrated in figure 83.

Recently, David Millen has written alternative communication routines which use the UDP and IP protocols [Millen94], and several researchers are using the device for tasks as diverse as analysing the chemical composition of cross-sections of human hairs and of the inside of the Joint European Torus plasma containment vessel.

7.1. Introduction

Micro-analysis using MeV ion microbeams is now well established [Watt87]. Particle-induced X-ray emission (PIXE) is often sensitive to elements in the target at the parts-per-million level, and is very useful for chemical analysis of microscopic samples in a variety of fields as diverse as archaeology and medical research. Rutherford Backscattering (RBS) of the analysis beam is widely used to obtain thin film chemical depth profiles in the material sciences. One micron diameter ion microbeams can readily be formed, giving very detailed three-dimensional information.
Conventional techniques for processing RBS and PIXE data rely on capturing and storing the information and processing it offline - [Takai92] showed how a CAMAC-based 16-bit CPU could be used to control the beam scanning process, and how a 32-bit Motorola 68030 was used to buffer up to 8 Mbytes of raw data before transfer to an Alliant vector processor for off-line analysis.

Rather than buffer data for subsequent analysis, we have constructed transputer-based data collection and pre-analysis equipment, which reduces the amount of storage and buffering required to manageable levels. The partly-processed results may be transferred across Ethernet to a Sun workstation for further processing, graphical display and archiving. The user is able to interact with a control panel window on the Sun's display, too, and can thus control what is recorded by the transputers, and from where on the sample, in near real time. This further reduces the data collection problem, because the initial sampling need not be so exhaustive.

By using a combination of transputer and Sun hardware, the front-end data collection may be performed by processors well-suited to high-speed interrupt-driven communication and timer-based operations, while the human interaction may be designed around X-windows displays which can be situated remotely from the ion beam laboratory.

7.1.1. Design Parameters

The objectives of the system can be summarised as:

- To scan the microbeam across the sample (in the X- and Y- axes), dwelling for a set period (typically between 100 µs and 100 ms) at each position, as shown in figure 79.

![Figure 79 - scanning the sample](image)

Apart from the planned dwell times, other pauses in this scanning process may not be tolerable, depending on the damage being done to the sample by the beam of ions. Scanning areas are typically 256*256 pixel squares, but smaller or non-rectangular areas can be required for some analyses.
At each (X, Y) beam position, to collect the outputs (which represent energy levels) from the RBS and PIXE detectors, to convert them to digital form using analogue-to-digital converters, and to update two types of data structure. One structure is a map which shows the number of detector outputs which fall between specified upper and lower energy thresholds. A 16-bit counter stored in the relevant (X, Y) position is simply incremented if the thresholds are satisfied. Several maps, with different thresholds, may be collected simultaneously, for different areas. Each map is stored as a two-dimensional array of INT16 values.

The second type of data structure is an energy spectrum. For specified areas of the collection plane, a 32-bit counter is incremented at the linear position in the spectrum which corresponds to the energy level detected. Several spectra may be accumulated simultaneously. Each spectrum is stored as a one-dimensional array of 512 values.

Detector outputs may be received at intervals no closer than 5µs (due to the signal conditioning electronics), although typical rates are only around 4000 per second.

To allow the user to specify overall scan areas, the areas over which to accumulate the spectra, and the range of energy threshold levels for each map. Normally, four spectra and four maps might be collected at once.

![Figure 80 - ion microbeam - hardware block diagram](image-url)
7.2. The Experimental Hardware

The basic elements of the system are shown in figure 80. Working from the ion beam and associated detectors back to the user interface, the important units are:

7.2.1. The Beam Line and Beam Steerer Units

The beam of ions is produced by a van de Graaff accelerator which typically generates a beam of 2 MeV Helium ions (for RBS) or Hydrogen ions (for PIXE). The beam passes through an object aperture with selectable diameters of 1 mm, 0.2 mm or 0.05 mm, and is then focussed magnetically, reducing its size by a factor of 5 in each dimension. See [Mynard85] for further details.

The positioning of the beam is done by two pairs of steerer plates, one pair for each of the X and Y axes. These are driven by a valve-based power supply and amplifier, from analogue control signals which originate in the transputer. The maximum controlled slew rate is 300 V/ms over the 10% to 90% central range of operation. The uncontrolled flyback time is much faster - about 1 \( \mu \)s. The map images shown in figure 81 represents an area about 1 mm square, and thus required steerer plate voltages of about 1 kV from side to side. They contains 128 lines, yielding a minimum frame time of 430 ms. A faster amplifier could easily be constructed to improve this.

These are map images of a copper test grid. Light colours indicate areas where there are more counts; dark areas generated fewer counts. The recording conditions were a 600 pA 1.75 MeV H+ beam, nominally focussed to 0.01 mm. The maps are 128*128 pixels on each side. The grid squares are 0.1 mm on a side, and each image shows an area of the grid about 1 mm across. The left image is of the copper X-ray signal, and the right is the total RBS signal. The images were obtained simultaneously in about three minutes.
7.2.2. The Transputer Interface

The interface between the transputers and the beam hardware is mapped onto two transputer links. These carry the XY positioning requests to the two 12-bit digital-to-analogue converters (DACs) which control the beam steerer plates. A third DAC is used to provide simulated data which can be fed back in place of that from the real detectors, allowing software testing without the need to use the beam. The 12-bit DAC values, together with a 4-bit identification field, are transmitted as INT16 values along one transputer link. The inputs from the two real detectors (one of which may be substituted by the simulated output) are fed to 12-bit analogue-to-digital converters (ADCs) and then to an arbitrator unit, which queues the inputs, appends a 4-bit code which identifies the source (with plenty of spare codes for expansion), and feeds them to the transputers via their links. The 16-bit values are transmitted as two 8-bit bytes, one along each of two transputer links. This is marginally faster than transmitting the two bytes sequentially along one link, because of the lack of overlapped acknowledges on Link Adaptor (IMS COl 1) links.

7.2.3. The Transputers

Three transputer TRAMs have been chosen for the operational version of this system, although various other configurations were adequate for earlier testing. The control of the ADCs and DACs, and the accumulation of received energy measurements, is controlled by a dedicated IMS T805 with 8 Mbytes of memory. This is interfaced to an IMS T805 with 4 Mbytes of memory, which performs the overall control of the experiment, buffers data transmission to the Unix display processes and supervises the TCP/IP connections to the third transputer. The third transputer is the IMS T222 on an IMS B431 Ethernet TRAM; this processor simply transmits packets onto Ethernet and queues them on receipt. The three TRAMs are mounted on a Eurocard motherboard (the DTR2 motherboard which is described in section 6.2) which also carries an EPROM bootstrap facility and a watchdog timer, allowing software for all three transputers to be booted from a file on the Ethernet network. This considerably simplifies software development. The whole transputer configuration fits into a volume of 220 x 100 x 20 mm, making it easy to accommodate in a screened case next to the one containing the analogue converters.

7.2.4. The Host Workstation

The user interface for the system is maintained on a Sun 4 workstation with a colour display, running SunOS and X-windows. In normal operation, the workstation runs three processes which take care of the user commands, the map displays and the spectrum display, respectively. The workstation may be located next to the accelerator, or far from it provided that the network connection between it and the transputers can carry sufficient traffic to make real-time operation possible.
An early prototype of the system simply plugged the detector links into a single transputer (on a B004 board) which in turn was plugged into a PC. Unfortunately, managing several concurrent tasks on the PC proved complicated, and the multi-processing Unix solution is far more convenient and flexible. Also, most of the other software in this experimental area has been written for Unix, and it was important to maintain consistency.

7.3. The Software Process Structure

The software processes which run in parallel and make up the overall system are shown in figure 82, below, and in figure 83.

Three Unix processes make up the user interface of this software. A tree of TCP and IP processes, as described in Chapter 4, run on the B431 TRAM and on the first T805 to provide three TCP streams for communication with the Unix processes. In the centre of the structure sits a main controller process. It manages the real-time line scanning and data collection process, which runs on the front-end T805, and also controls an output buffering and routing process which maintains the user displays. All of the processes which run on the transputers are written in occam, whilst the Unix software is written in C.

The sequence of operation is that the networked transputers initially listen to the Ethernet for an incoming service request from a control-panel process, which runs on a workstation authorised to
access the microbeam. The Unix control-panel process is then informed of the TCP port numbers of the spectrum and map display channels, and it spawns display processes which make their own connections to the transputers.

![Diagram of hardware and software process structure](image)

*Figure 83 - the hardware & software process structure*
The control panel window contains several pull-down menus which allow the user to set up the number of maps and spectra required, and to specify their dimensions and energy level ranges, as necessary. It also allows the overall scan size and the time to spend listening for counts from each pixel (the dwell time) to be configured. All this control information parameterises the subsequent data-collection phase. Figure 84 shows the control panel window, with a pull-down menu exposed. All the user interface design and X-window coding was performed by David Millen.

When all the run-time parameters have been specified satisfactorily, the user may select the “Go” icon, and the control information is transmitted to the transputers.

![Control Panel Window](image)

*Figure 84 - the control window, showing a pull-down menu which enables the scan parameters to be specified.*

Once in run mode, the software on the front-end transputer RETYPES its memory buffer to the dimensions of the maps which are required, and zeroes its storage arrays. The controller transputer then issues a sequence of line-scan commands, and the front-end processor performs the scanning and data capture. At the end of each line, updated results are passed back to the output routing process, where they are buffered. Data can be transmitted back to the map and spectrum displays on the Sun (see figures 81 and 85, respectively) in parallel with collection of the next line of real-time data. If the size or complexity of the display data is too great for the network bandwidth or the screen updating rate on the Sun, the buffer can be operated in overwriting mode, so that the user sees complete updates rather less frequently than the scanning software generates them.
Compression techniques might also be used to reduce the communication requirements, but this would generate extra processing loads and necessitate another transputer.

![Spectrum Display](image)

Figure 85 - the spectrum display, showing test data.

Normally, a scan runs to completion and the whole system returns to an idle state. At this point, further data may be accumulated into the initial set by re-running, or the control panel parameters may be changed ready for an improved run, by specifying different energy thresholds or a different scanning area or zoom factor. By using three Unix processes, with all communication between the control panel and the others via the transputers, it is easy to act upon a press of the “stop” control button at any time - the control process on the transputer simply stops sending line scan requests and pumps a “poison” message (see [Welch89b]) around all of the scanning and buffering processes to bring them to a quiescent state.

Although the experiment is controlled via interactions with the control panel window, the spectrum window also has mouse-sensitive areas which allow the user to scroll and zoom around the display and to select which spectra to show. These actions are purely local to the Unix spectrum display process, and involve no communication with the transputers.

7.3.1. Strategy for Deadlock Avoidance

The process structure shown in figure 82 was specifically designed to keep deadlock issues to a minimum and to simplify the termination scheme which was mentioned in the previous section.
The process structure, with one exception, is built up from unidirectional logical message paths, several of which are then multiplexed onto the Ethernet which connects the controlling workstation to the transputers. Messages from the control panel process all propagate to the command interpreter process, and from there to the other processes. Results flow only to the spectrum and map processes, from which they are transmitted to two separate display processes on the workstation. The overall design, therefore, is made up from several simple pipelines of client/server processes (see section 2.3.2 and its references). Deadlock freedom, and simplicity in termination, follow directly.

The only bidirectional connection shown in figure 82 operates in client/server mode, with the control panel process acting as the client. This is also intrinsically deadlock-free. A future enhancement, which has been requested, requires the control process to reflect the status of the experiment via several icons - such as whether the scanning subsystem is running or stopped. This would involve the construction of the Unix equivalent of an overwriting buffer and a demultiplexer process on the logical channel from the transputer's command interpreter process back to the control panel process, to extract the asynchronous status updates and to maintain the most recent versions.

7.4. System Performance

The system has been designed with plenty of operational flexibility. Although per-pixel dwell times of around 10 ms may be normal, it is advantageous to be able to speed up the scan rate when positioning the beam and the target initially since the beam scan area can be seen visually if it is scanned fast enough. In addition, some samples may require their exposure to the beam to be minimised. For a frame time of 1 second, a scan of 1 mm × 1 mm and a pixel size of 5 microns (half the beam diameter), we require a dwell time of 25 µs per pixel. The front-end transputer can execute around 400 instructions in this time, sufficient to perform calculations for several events. The detector electronics are unable to present input counts at rates faster than 5 µs, and the ADC arbitration logic slows this slightly, so the front-end transputer can keep up. The code on this transputer is sufficiently small that it fits into the internal memory of the T805, maximising performance and eliminating the effects of fetching instructions from external memory. The large data arrays are held in external memory, of course.

With line scans taking a minimum of 200 × 25 µs in this example, each map display line is created in 5 ms. For a display with 4 maps, where each pixel is an INT16 quantity, the buffering and networking code must be able to transmit 4 × 2 × 200 bytes (plus a few address and tag bytes) in 5 ms. This rate - the fastest likely for meaningful data capture - therefore results in a network data rate of slightly more than 320 kbytes/sec. This is easily manageable by the TCP/IP code and hardware configuration in use, on which a rate of over 750 kbytes/sec has been measured. It also compares very well with the fastest throughput available from standard plug-in cards for the PC,
which are usually only capable of between 200 and 350 kbytes/sec and would, at best, be very stretched in this application.

Unfortunately, such data rates must be handled very carefully from within X-windows applications, and the designer must bear in mind the other processing and data buffering which the workstation is involved with. The major problem is that the main polling loop within X-windows applications seems to have a substantial latency, and thus if small network packets are each dealt with separately, the throughput drops. This effect is counteracted by passing map data to the Unix host in packets of up to 2 kbytes. These packets are filled with up to 1024 16-bit values from the contents of as many maps on each scan line as possible. More seriously, the XPutImage primitive which re-displays a window after updated data has arrived at the workstation is slow - about 70 ms for a group of four 128 x 128 pixel maps. It is thus necessary to restrict calling this routine to just a few times every second. Special command packets, passed across the network from the transputer controller process to the map display process, orchestrate this. To the user, maps evolve in a series of jumps rather than smoothly, line by line, but the animated effect is perfectly sufficient for experimental control and for evaluation of the data.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Function</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Run-time looping and timing overhead</td>
<td>9.5 μs per pixel</td>
</tr>
<tr>
<td>(b)</td>
<td>Map setup time</td>
<td>3.4 μs per map pixel, per frame</td>
</tr>
<tr>
<td>(c)</td>
<td>Map transmission time</td>
<td>4.9 μs per map pixel, per frame</td>
</tr>
<tr>
<td>(d)</td>
<td>Spectrum setup time</td>
<td>550 μs per spectrum (no memory); or</td>
</tr>
<tr>
<td>(e)</td>
<td>Spectrum setup time</td>
<td>0 μs (large memory overhead)</td>
</tr>
<tr>
<td>(f)</td>
<td>Spectrum transmission time</td>
<td>4700 μs per spectrum</td>
</tr>
<tr>
<td>(g)</td>
<td>Data accumulation</td>
<td>5 μs (variable) per sample received</td>
</tr>
</tbody>
</table>

Table 10 - costs of collecting maps and spectra

7.4.1. Performance Evaluation

Although the communication rate is sufficient for the current experimental work, investigation of the communication throughput has been considered important. The worst-case performance will be seen when sending many small maps, since the number of overhead calculations and message headers is largest in this case. A number of trials were performed to confirm this, generating 128 x 128 pixel maps and propagating them from the front-end ADC collection process to the Unix
display screen. By reducing the scanning dwell time to zero, the inner ADC collection loop could be forced to terminate immediately it was entered, ensuring that the whole system free-ran at the maximum scan rate possible. Although all the loop code and the termination mechanism was unchanged, the execution time of the transputer's talwt and alwt instructions in the main ALT construct (which listens for the end of the dwell time) was reduced by about 2 µs from the time which they would take when running the scanning system normally [INMOS89a]. This is because the ALT's requested termination time was always the same, or earlier, than the time that it was entered. The 9.5 µs figure in Table 10 above could rise to about 11.5 µs as a consequence.

Once the inner loop was free-running as fast as it could, the incremental expense of requesting different numbers of maps and spectra could be found by subtraction. Choosing to collect no maps and no spectra exposed the relatively high costs of scanning the beam and running through the per-pixel loop termination scheme, without the initialisation and thresholding cost of the analyses. The costs of including each element in the data collection scheme are summarised in Table 10. The two spectrum setup times ( (d) and (e) ) refer to whether each spectrum mask is re-calculated per line, or pre-calculated and stored in a table (which will be as large as a map if it is to be accessed fast).

This analysis shows that four maps and four spectra may be collected for a 200 x 200 pixel sample with an overhead of 23.1 µs per pixel - ( (a) + 4x(b) ). The cost of accumulating data values (g) is minimal at typical rates of 4000 per second. The time taken to transmit the results (c) is high, but may be completely overlapped with the setup and data collection for the next scan line.

Increasing the number of copies of the map data transmitted at the end of each line scan allowed the transmission path to be loaded until it saturated, without affecting the workload of the front-end processor unduly. This showed that the combined effect of controlling the front-end data collection, multiplexing three TCP/IP streams, performing all the protocol processing and buffering messages to the B431 Ethernet TRAM saturated the middle transputer in the pipeline at an effective data rate of 434 kbytes/sec. This was comfortably higher than the maximum rate for the entire application which is estimated earlier in this section. Subsequent experience of modifying the layout of each transputer's memory map, to optimise the use of on-chip memory, leads me to believe that over 500 kbytes/sec should be attainable. Running the links which interconnect the transputers at 10 Mbits/sec or 20 Mbits/sec made almost no difference to the measurements.

At network data rates of 434 kbytes/sec (mainly comprising map data), neither a Sun IPX workstation nor a Sun SparcStation 10 was overloaded with displaying the images, provided that computation on, and movement of, the transmitted data was kept to a minimum by sending it in the most natural order. To this end, one reason for the high cost of (b) in Table 10 is due to the extra indexing operations needed to read the maps by column, rather than by row. "High cost", is relative, of course, and refers to each of many of tens of thousands of pixels in a map. Pipelining the TCP/IP code onto two transputers and separating the experiment controller process onto a third would make it possible to increase the data rate to around 850 kbytes per second. This would bring
the less-powerful Sun IPX close to saturation, if equally-hungry display routines continued to be used.

The tuning of the system described, and the measurements above, were complicated by the difficulty of identifying which elements of the overall system were under the heaviest load. The loose synchronisation provided by TCP/IP's windowing mechanism ensures that the Unix processes can as easily delay the transputer application as vice-versa, and the difficulty of extracting diagnostic information from a cluster of transputers booted from, and only attached to, a local area network needs to be addressed.

7.4.2. Future Work

One aspect of the design which warrants further investigation is the accuracy of timing of the per-pixel dwell interval. Although this code is already running alone in the front-end transputer, extra buffering and further consideration about where to perform the scan mask and thresholding calculations might ensure that the dwell interval is not disturbed by a few microseconds if a received data value should be being processed as the dwell timer expires. This effect in the current implementation may be discounted on statistical grounds, however.

After initial experiences of controlling experiments and of viewing the results, most of our other requirements centre around graphical, rather than numerical, methods for specifying and changing the scanning parameters. Individual contrast controls for the various maps are also necessary so that minor variations in energy density can be more easily identified. The ability to specify irregularly-shaped areas of the target, and later to scan them and to accumulate spectra for them at the front-end event rate would be useful, too.

7.5. Summary

The system presented in this chapter is operational and is collecting valuable results. The benefits of using transputers for the front-end data collection are obvious - their performance is well-matched to the physical limitations of the system, and can be tailored by the addition of extra processors. In addition, the division of the task between transputers and a Sun workstation has ensured that the user interface software could draw upon X-windows routines and programmer experience, whilst the real-time scanning could also be performed on an appropriate architecture. The message-passing programming method is perfectly applicable to the networking connections, and the communicating process architecture of the transputers extends neatly to the three Unix display processes. The use of occam for the transputer software made that particularly simple to design and to implement (a workable demonstration took about two man-days to write and to couple in with already-complete X-windows graphics code), and the simple structure of separate command and display processes again sub-divided the task in a convenient manner. My TCP/IP
software, which connects the two regimes, proved stable and easily capable of supporting the message traffic generated.

The conclusions of detailed architectural and performance evaluations are that:

- Two transputers were required to obtain a good timing resolution and sufficient preprocessing performance.

- Using three TCP streams to connect the Unix workstation to the transputers considerably simplified the Unix display software, since each type of display window could be handled from a separate process which did not need to know about the presence of the others. This made initial testing of the display processes independent of the integration of their networking software calls and final testing against the networked transputers.

- The fastest network transmission rate required by the scanning application (320 kbytes per second) was easily achievable by the hardware configuration used, which was measured to be capable of 434 kbytes per second when the application timing delays were removed.

- Measurements of the experimental hardware, but with one more processor added to carry the IP and TCP root routing processes, showed that this slightly extended architecture would support over 850 kbytes per second (more than double the maximum required). In these circumstances, the workstation running the display routines appears to be the factor which limits the throughput.
8. Continuing Work and Future Perspectives

This thesis describes research which justifies the design of a suite of parallel protocol-processing software and explores the limits of its performance. There are, of course, many issues which have not been covered in the time available; this section highlights three of those which are worthy of further examination.

8.1. Extra Parallelism in the TCP Process

The TCP Process is currently implemented as a large central server process, surrounded by a number of satellite parallel processes which buffer incoming and outgoing data, and harness two shared memory buffers - see section 4.1.5.

Recent attempts to split the TCP process into parallel receive and transmit pipelines have proved complicated, although the task appears to be tractable and is considerably aided by the strict declaration, type checking and protocol checking requirements of the occam programming language.

The main effect of separating the TCP process into two pipelines is that there are several variables which need to be known to both halves of the process - in particular, those which define the transmit and receive windows. In a split-process implementation, updates to each window are made from both pipelines - the head pointer by one side and the tail pointer by the other. Current indications are that the cost of exchanging these values sufficiently frequently to prevent the windows filling up (or running dry, from the viewpoint of the TCP process at the other end of the connection) will balance the gains from splitting the TCP process in the first place. The exchange cost is due to the communication from one part of the split TCP process to the other each time that a network packet is processed; this involves two channel communications via an overwriting buffer process, to eliminate deadlock (see figure 86).

Compared with a single ALT invocation per packet in the original process, both the new split processes execute an ALT construct for each packet processed. The combined number of ALT guards in the new case is the sum of the number in the original process, plus two extra ones associated with the channels involved in the transfer of the state variables. Since the execution costs of the ALT constructs are a major part of the overall TCP processing effort, we can expect the overall load to increase in relation to the number of guards processed. The increase is mitigated by the opportunity for the new process structure to deal with incoming and outgoing TCP packets in
parallel, reducing TCP message latency, and by the controlled overwriting of the window pointer values when the workload becomes high, which reduces ALT invocations. It is difficult to predict whether performance will be improved as a consequence of these modifications. This exercise is therefore worth continuing; even a negative result will be academically valuable.

8.2. T9000 Issues

The design of this TCP/IP software pre-dates the availability of the new IMS T9000 transputer. This section contrasts the current architecture with an enhanced version which could use the new features of the T9000 transputer and of the IMS C104 packet router.

The existing software was designed around the constraints of the relatively low link throughputs of the first-generation transputers and of their restricted link connectivity. Thus, inter-processor communications only run at up to 20Mbits/second, minus overheads, and multi-processor configurations are confined to simple structures such as arrays and pipelines. A corollary to these parameters is that there is no need to separate user data from its control headers throughout the software pipeline, as demonstrated by the performance of my software (discussed in section 4.4), in contrast to shared-memory methods such as the INMOS and Karlsruhe architectures which were introduced in section 2.6. In principle, these restrictions are removed by exploiting the T9000 and C104 products.
The T9000 transputer increases the bit rate of its links to five times that of the first-generation transputers, although the lower overhead of the T9000 acknowledgement mechanism yields gains of up to 5.5 times for unidirectional traffic and 7.0 for bidirectional traffic, assuming message sizes greater than about 20 bytes [INMOS89a], [May93, chapter 6], [Hipperson93]. Software through-routing of messages, for processors which are not directly attached, is eliminated in the T9000 architecture by provision of a Virtual Channel Processor (VCP) on each T9000, and a separate routing device (the C104) which utilises wormhole techniques to pass messages between the VCPs of each processor in a network. Not only does this considerably reduce the expense of communicating across multi-hop paths, effectively making each path just a single communication, but it also introduces freedom into the placement of processes, and therefore eases load-balancing.

Matching the increases in communication throughput (between a T805, say, and the T9000) is a corresponding increase in processor performance, although this is partly achieved by a number of non-deterministic methods such as caching (of memory and workspaces) and by pipelining, and is therefore not guaranteed. An enhancement to the communication paradigms of the first-generation transputers is that of shared resources, in particular of shared channels. These permit a T9000 server process to accept requests along a single channel from a number of client processes, queueing them (in the order of receipt) until it can service them. Acceptance of messages along shared resource channels requires far less processing than the set-up and clear-down of guards to conventional alternation structures. Shared resource channels may be used to construct multiplexers and similar simple processes, and also to provide complex alternation constructs in places such as the central state machine of my TCP process.

The effects of the new T9000 architecture on my TCP/IP design are therefore many and varied. Perhaps the most significant effect is that virtual channels remove the need for the TCP router processes to fan out the stream of IP packets to the separate TCP processes, since each TCP process may be connected directly to the output of the IP process. The function of the root router process (which decides which TCP process each IP message should be sent to) is still required, and this could remain separate from the IP process, as at present, or could be combined with it. TCP packet checksumming must also be retained at this level, and a number of checksumming processes (corresponding to the remainder of the processing in the current TCP tree router processes) will therefore still be required. These processes could be statically placed on each TCP channel, or could be accessed from a common pool using the process-farming concept - see figure 87. Checksum calculation in a process-farm becomes viable because the cost of sending IP packets to a checksum farm process is only that of a single data communication, and the checksummed packet may then be forwarded directly to the relevant TCP process in another communication - the same as that of a packet progressing through a single level of the TCP router tree. Additional small communications are required to determine a free worker process for the operation, but these occur after each packet is processed and should not affect the message latency. All of the arguments above apply equally to outgoing TCP packets travelling towards the IP layer.
Another variation which becomes possible with virtual channel routing is that a single TCP process could be used, in place of the separate TCP processes which are used for each stream at present - see figure 88. A single TCP process would have to identify the stream to which each incoming packet belonged, and manage all the state for that connection using an index to its part of the overall connection state space - rather closer to the Transmission Control Block (TCB) mechanism described in the TCP specification [RFC793 - Postel81c]. This identification work is currently done in the TCP root router process. The indexing is extra processing effort.

Using a single TCP process only becomes viable when distribution of the TCP streams to the user application processes is performed along virtual channels, because otherwise trees of application router processes would be required. In the application-to-TCP direction, messages may be received by a single TCP process from many application processes relatively cheaply using the new shared resource channel mechanism. In addition, using the ignorant server technique [INMOS93a, chapter 12] in the single TCP process would allow any number of user applications to connect to the TCP process at run-time, with no need for prior compilation or configuration of their channel connections. A difficulty with the concept of a single TCP process is that having just one such process eliminates the possibility of load sharing. Adding it, using process-farming, would require
all the state variables for each connection to be stored centrally and to be distributed to the relevant TCP process together with each work request. Provided that the static allocation of one TCP process to one TCP stream remains acceptable, there is no need to move away from the current process structure.

One of the main foundations of my design, which differs from those of the Karlsruhe [Zitterbart91], IBM Zurich [Rutsche93] and UNSW [Chan94] architectures, is that I have avoided shared memory and have passed all the message traffic along the protocol-processing pipelines, rather than storing it in a central buffer pool using shared memory techniques. By scaling my measured results to T9000 processors, a fully-pipelined approach should allow throughputs of around 5 Mbytes/second across a suitable network to similarly-performing peer processors. If it were possible to separate the packet data from the packet headers, I have demonstrated packet processing rates of about 3000 packets per second using T800 technology, which could be expected to scale to 15,000 packets per second on T9000s - similar to the measurements and projections of the other authors listed above. It is therefore possible, in principle, to carry TCP packets of over 1400 bytes at this rate - 21 Mbytes/second or nearly 200 Mbits/second at the network. These rates depend on being able to communicate such a volume of data within the protocol-processing architecture. On T9000s, 21 Mbytes/second requires more than two links running flat out, concurrently, since each is only capable of 9.55 Mbytes/second [May93]. Even using the T9000's grouped adaptive routing facility - by which hardware links may be treated in sets and traffic be shared between them automatically - these data rates are impossible to achieve using a directly-connected pipeline with two incoming links and two outgoing ones. It therefore makes sense to use several T9000s as a data repository, and to load-balance data across them, checksumming it on the way, as shown in figure 89.
This is very similar to the IBM Parallel Protocol Engine [Rutsche93], but uses transputer link communications in place of shared memory. Of course, the checksumming could be done in the link-layer hardware, rather than in software, although the summing workload is not excessive for T9000s at the data rates being discussed. One difficulty which these schemes share, in contrast to the simple pipeline, is the need for the processes to send requests to the central repository before the stored data can be returned; the latency of the associated packet search function and return data transmission may be unacceptably high - as seen in the INMOS TCP/IP architecture. Some of this latency may be masked by specialised hardware, but this would increase costs and reduce the benefits of a T9000 software-only solution.

In summary, the main benefits of the T9000 architecture to layered protocol processing appear to be from its virtual channel mechanism and from the shared resource channels which this facilitates. In particular, virtual channels would support a revised design for the distribution of packets, within a streamlined TCP routing scheme. Additionally, virtual channels make a central shared packet store easily accessible from multiple transputers, and this opens up the possibility of splitting the packet headers from their data payloads, processing each separately and not allowing the data to interfere with the communication of the headers along their pipeline.

8.3. Resource Allocation in the Networked Transputer Facility

One subject which has not yet been tackled (and which could be dealt with more simply by the T9000 / C104 combination) is how users of the networked transputer facility could bid for, capture and configure a number of target processors from a pool made available for multiprocessor target programs. The necessary mechanisms are fairly clear. Many are discussed in [Accetta83 - RFC8873], which defines a broadcast-based protocol for distributing resource requests across a distributed architecture. Resource manager processes should be placed in centrally-accessible locations - the root router processes of each protocol stack in my first-generation accessible architecture - where they can receive requests from the network and transmit configuration information to each of their subsidiary client Iserver processes. This is shown in figure 90.

A remote session using more than one target transputer will operate similarly to one which uses a single transputer, except that negotiation for use of the pool of target processors will have to take place prior to loading the target code, and the extra processors used will have to be returned to the pool once the session is finished. A negotiation will start when the resource manager process receives a request from the client Iserver; this will specify the target processor interconnection arrangement. The resource manager will attempt to build this configuration of processors from those in the free pool, and (if successful) will connect the target’s boot link(s), reset, analyse and error lines to the requesting Iserver’s original target transputer, which acts as the host processor for the new network, and provides a location to run loader and debugging tools.
At the end of a service, when the target transputer configuration ceases running the application program and any subsequent debugging sessions, a simple message from the client Iserver back to the resource manager allows the idle processors to be returned to the free pool. As with any resource locking activity, notice that deadlock may occur if two Iservers should request configurations which cannot be built together, and some of the free pool of processors are reserved for each before all the required processors become available. This problem may be resolved by a more sophisticated algorithm in the resource manager process, but care must be taken not to introduce polling in the interfaces between the user, the client Iserver and the resource manager. Deadlock avoidance in this situation is a conventional operating system issue, and is discussed in [Peterson85, chapter 8], amongst many others. Methods for interconnecting groups of transputers are discussed in [Nicole88]. Notice that both of these issues may be computationally expensive.

This resource allocation scheme is not unlike that proposed in [Kerridge94b] for a similar resource sharing issue in a T9000 processor farm. Obviously, the T9000 version can use virtual links, connected by the resource manager, rather than the more restrictive hard links. The higher overheads of dynamic task creation in [Kerridge94c] are less applicable in this context.
9. Conclusions

Motivated by the need to access transputers across an Ethernet local area network, this thesis has shown how a TCP/IP software suite was designed and evaluated. Considerable amounts of parallelism have been incorporated into the layered protocol processing algorithms and into the inter-transputer data stream buffering. Apart from the core software implementation, several other activities have supported this research, including hardware design and the creation of two example applications.

The central TCP/IP processes are suitable for a number of tasks:

- General-purpose networked transputer facilities may be constructed, such as the networked Iserver described in chapter 5. These devices should be flexible in terms of size and configuration; they should be economical and they should offer high performance. Given the limitations of Ethernet itself, the networked Iserver satisfies these aims by being able to support any reasonable population of target transputers, using a tree-structured packet distribution arrangement. It also offers file access performance which can match that of the best PC-hosted facilities, as well as providing high-performance hooks into the underlying TCP/IP protocols for network access from other processors.

- Transputer-based TCP/IP code may be incorporated into stand-alone embedded systems and, like the transputers themselves, it is particularly suitable for instrumentation and control purposes. Raw performance is one of its important benefits in this role, especially as its throughput characteristics can be tailored to the application by altering the number of transputers in the pipeline. Other benefits are that revised versions of the embedded system software may be loaded from boot files stored on the network, and (conversely) that the architecture supports remote dumping of these transputers. Both of these aid development and debugging of the application code.

- Hybrid designs, in which occam programs are run partly on transputers and partly on non-transputer hardware (such as SPARC, Power PC and DSP processors), are well-supported by the all-transputer architecture with its high network throughput. This could enable the efficient hardware interfacing and low-level real-time software support of transputers to be coupled with the greater processing capabilities of networks of modern workstations, all packaged in a single occam program.
The design of the TCP/IP software, as a number of parallel processes, mirrors the layering of the protocol itself. In addition to the TCP and IP processes, checksumming and distribution of packets is carried out by an extensible tree of TCP routing processes. This provides an economical mechanism for attaching many transputers to the Ethernet, since adding an extra layer to the tree of transputers multiplies the number of leaf nodes by three. Each time that the tree is extended, the new intermediate processors may be configured to perform a share of the checksumming duties. Sadly, it has been shown that the protocol-processing processes on each transputer must share their memory arrays with the buffer processes which handle transmissions along the transputer links if optimal performance is to be obtained. Since this contravenes the occam language's aliasing rules, although in a manner which can be demonstrated to be secure, language extensions to enable the use of pointers in these special scenarios would be desirable.

Examining and refining the performance of the basic TCP/IP code and of the example applications has formed a large part of the work reported in this thesis. Section 4.4 shows that each of the individual TCP/IP protocol-processing modules has a peak throughput of over 1400 kbytes/second. This is usually only achieved when passing large data packets, and tends to be constrained by the data rate of the transputer links and by any inter-packet delays caused during processing. As the packet size falls, the number of packets processed per second has to rise to maintain the same traffic on the links, and the constant costs of dealing with each packet eventually increase the processor load to saturation. These characteristics have been explored. Careful attention to the placement of critical code modules in the transputers' on-chip memory was vital to give consistent timings and high throughputs - a problem which became worse as more processes were combined on the same processors.

Following measurement of the individual processes, they were combined into groups which each implemented complete TCP/IP stacks. These stacks of processes were configured in five different ways in order to run them on differing numbers of 20 MHz transputers, as shown in figures 61 and 66. Initially, two such stacks were run back-to-back, through another transputer which simulated the intervening network hardware. Maximum unidirectional throughputs of 1181 kbytes/sec, and bidirectional rates of 1223 kbytes/sec were measured when using four transputers in each stack. These rates decreased only slightly when three transputers were employed on each side, and dropped to a worst case of 583 kbytes/sec when all the protocol processes were run on a single transputer.

Measurements were also made of each protocol stack when it was attached to an Ethernet-equipped transputer which was connected across the network to a Sun workstation. End-to-end throughputs ranged from 576 kbytes/sec when running the protocol processing on a single transputer - corresponding closely to the 583 kbytes/sec figure above - to 960 kbytes/sec with four transputers in the stack. Again, three transputers delivered a performance which was close to the maximum value obtained from four transputers. A marginally higher throughput - reaching 976 kbytes/sec - was
obtained by substituting 22.5 MHz transputers in place of the slower ones which were running the IP and TCP routing processes.

In the more capable multi-transputer configurations, the throughput of the protocol processes appeared to be restrained by the IMS B407 Ethernet interface. On investigation, this module was found to be unable to transfer more than 1096 kbytes/sec from a transputer link onto Ethernet and 1072 kbytes/sec in the opposite direction, due to interference between the DMA operation of its Ethernet controller chip and the memory accesses of the transputer links. Allowing for the TCP and IP packet header overheads, the four-transputer protocol stacks came to within 4% of these transmission rates, which is considered to be highly acceptable.

Two example applications which use the parallel TCP/IP software are reported in chapters 5 and 7. These examples demonstrate the utility of the networking software in non-trivial circumstances, and indicate some of the design techniques which have been found to work well when incorporating it.

Normally, transputer programs access filestore, the screen and the keyboard in a host computer to which they are attached by client-server interactions which accord with the INMOS SP link protocol specification. The networked Iserver propagates file requests from one of a bank of target transputers, across Ethernet to a host workstation where they are serviced, and, for each request, returns a response message. By constructing the network side of this server entirely from transputers, it has been possible to exploit the high throughput rates of the parallel TCP/IP code, as well as the simplicity of the multi-transputer hardware architecture. It is possible to construct such a networked facility capable of serving as many target transputers as desired from the kit of hardware and software components which have been developed. File access rates in excess of 250 kbytes/second have been measured, even allowing for the high latency of the network connection. This is similar to the service provided to transputers by a fast directly-connected PC host.

An example of a networked embedded system has also been developed. This enables a control process, running under X-windows on a Unix workstation, to send commands to an all-transputer based ion microbeam instrument. On receipt of commands, the instrument configures its local memory, sets up a scanning regime and initiates the collection of image and spectral data from sensors which pick up energy dissipated from an object when it is bombarded with a beam of ions. Time-critical activities are performed on a front-end transputer, which pre-processes the received samples away from any influence of the network. A second transputer passes results back to two display processes which run on the controlling user’s workstation. This transputer is easily capable of sustaining the maximum data rate of the application - currently 320 kbytes/sec - and could be augmented with one or two more transputers if higher rates were needed. Complete image frames may be overwritten in a controlled manner if the network or the receiving workstation are found to be incapable of these transmission rates.
Chapter 8 introduces a number of topics for future study. One of these examines the applicability of the pipelined TCP/IP software architecture to the new INMOS T9000 processor. It concludes that the basic process decomposition is satisfactory, and that the new T9000 virtual channel routing mechanism could replace the existing packet distribution function of the TCP router processes. In addition, the associated shared resource channel mechanism could reduce the cost of alternations everywhere in the code and allow process farming to accommodate the checksumming activities of the TCP router processes. Data rates would ultimately be limited by the T9000 link throughputs, even if grouped adaptive routing were to be adopted.

In conclusion, I have successfully demonstrated the value of occam and transputers in designing and developing a low-cost, high-performance distributed communications architecture.
10. REFERENCES


G. Jones : "Efficient multiple buffering in occam", in the Occam User Group Newsletter, no. 11, pub. INMOS Limited, Bristol, July 1989, pp. 36-44.


[various RFCs] Many "Request for Comments" (RFCnnn) documents published by the Information Sciences Institute, University of Southern California, 1981 - date, under contract to the U.S. Defense Advanced Research Projects Agency, which form the TCP/IP specification. RFCs are obtainable by FTP from src.doc.ic.ac.uk in directory /public/computing/internet/rfc.


