A File Server for a Multi-Transputer UNIX System

P K Hoffman and G de V Smit
Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

The DISTRIX operating system is a multiprocessor UNIX-like distributed operating system. It consists of a number of satellite processors connected to central servers. The system is based on the MINIX operating system, which is in turn based in UNIX Version 7. A remote procedure calling interface is used in conjunction with a system wide, end-to-end communications protocol that connects satellite processors to the central servers. A cached file server provides access to all files and devices at the UNIX system call level. The design of the file server is discussed in depth and the performance evaluated.
Keywords: *File servers, multiprocessor system, operating systems, satellite systems, transputer.*
Computing Review Categories: *C.1.2, C.2.4, D.4.3, D.4.4*

Presented at Vth S.A. Computer Symposium

1 Introduction

The DISTRIX project, undertaken by the Laboratory for Advanced Computing in the Department of Computer Science at the University of Cape Town involves the research and development of a distributed version of the UNIX[1] operating system for a network of transputers [14].

The goals are, amongst others, to investigate the feasibility of UNIX on transputers, to gain experience in the development of distributed operating systems and to produce a workbench for future experimentation.

In this paper, the design of the DISTRIX file server and its environment are discussed. Foremost in this environment are the communication mechanisms that inter-connect the modules of the file server and connect the file server to the rest of the system.

In the UNIX environment, both the software controlling the files and the disk image itself are often referred to as "the file system." In order to avoid confusing the two, we will refer to the former as the *File Server* or FS. It has the functionality described in Birrell and Needham [3]: "*...having as its main purpose the storage and retrieval of bulk data.*"

2 Overview of MINIX

To simplify the development of a prototype system it was decided to use MINIX [16] as the starting point for the operating system. Motivating factors were the availability of unlicensed source, the relatively small size, and the modular design of MINIX.

Functionally, MINIX is similar to UNIX Version 7 (V7) [2], providing a full hierarchical file server, most of the V7 system calls, and a multitasking environment in which to run applications. It is written for the IBM PC range of microcomputers, based on the Intel 80x86 microprocessors. The shell is based on the Bourne shell [4] and includes such script language features as were available in V7.

The internals of MINIX differ greatly from conventional UNIX systems. A UNIX system kernel is generally one large, monolithic program performing system calls by means of kernel traps. Conversely, the MINIX kernel is structured as a number of discrete processes connected by means of a message passing layer. System calls are effected by means of this message passing mechanism.

The modular structure of MINIX is illustrated in Figure 1. The servers are completely independent, discrete processes. They communicate only by means of the underlying message passing mechanism. Both server processes operate as user level processes, but have privileges not granted to normal user processes. Within the system software itself, the kernel (responsible for process scheduling and providing the message passing layer) is connected to the memory manager and file server by this same message passing mechanism.

This layered, modular design has resulted in a system that potentially lends itself well to distribution amongst separate processors connected by a hardware message passing mechanism.

3 Overview of DistriX

Figure 2 illustrates the prototype system architecture. Each block (except the prototype disk and terminal devices) represents an Inmos transputer [8]. The

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

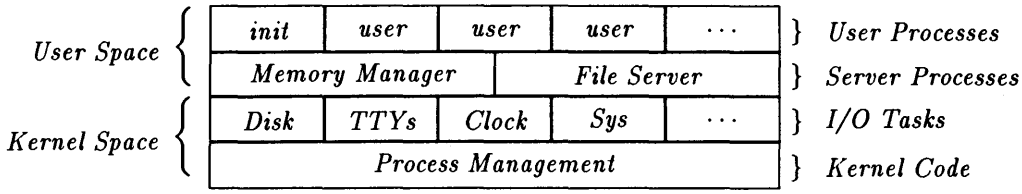

Figure 1: Structure of the the MINIX operating system

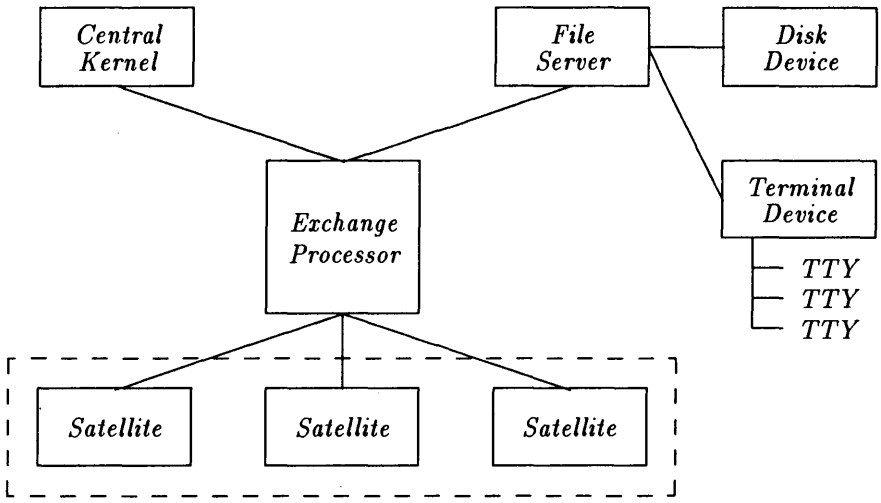

Figure 2: DISTRIX prototype system architecture

transputer is particularly suitable because of its high speed (around 10 MIPS), simple interprocessor communication (using built-in hardware links), machine instruction for blocked message passing along channels and a microcoded process scheduler.

In the prototype, transputers are hosted by a number of PC/ATs. The PC/ATs are also useful to attach devices such as disks and terminals as well as for downloading the operating system image at boot time.

In adapting MINIX for this transputer-based, multiprocessor environment, a modified version of the satellite model [1] was used. This model best suited the requirements of the proposed system [14]. All user processes are executed on the satellite processors. Each satellite executes a subset of the MINIX kernel called the mini-kernel (MK). When a user process (client) issues a system call, the call is passed on to the MK where one of three actions is taken:

1. the call is serviced fully by the MK,
2. the call is passed to the central kernel (CK) for processing, or
3. the call is passed to the FS for service.

All file system calls are always passed on to the FS as the MK does not carry out any such calls.

The rest of this paper discusses the provision of a system wide communications protocol to act as a message passing mechanism and the relocation (to a separate processor) of the file server. First, an overview of the exchange is given.

The Exchange

In DISTRIX, the various components of the system are on separate processors. In order to support a global message passing system, every node must be able to communicate with every other node in the system.

A transputer only has four external links to connect it to other transputers. It is possible, however, to increase the number of available links by memory mapping link adapters [8] onto locations in the transputer's upper memory, thus allowing more than the standard 4 bidirectional links to be used. These are referred to as *pseudo links.*

Such modified transputers, together with simple routing software to create a switching exchange, are used in the DISTRIX system. Input links on the exchange transputer are polled for message packets which are routed to the appropriate output link depending on the destination address in the packet headers and a local routing table. Each input link has its own dedicated buffer, thus providing a store-and-forward, fully interconnected network.

It is possible to cascade multiple exchanges to connect more processors than are allowed by a single exchange. Furthermore, separate exchanges may be introduced allowing separate paths to the File Server and Central Kernel, potentially halving the load.

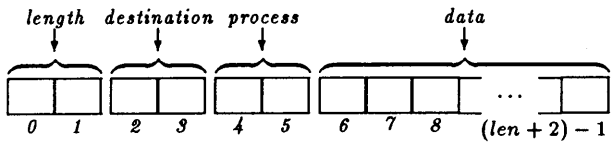

Figure 4: DISTRIX *message format*

The maximum message packet size is 64KB. In reality, the exchange only has sufficient memory for 8KB buffers. The protocol emerges as efficient for long messages, with a header overhead of approximately 0.6% for a 1KB packet.

The higher protocol layers communicate with their respective peers on other processors, oblivious of the medium below them, and independently of any changes that may be applied to the lower layers. The four layers are outlined in the following sections.

Physical Layer (0)

The physical layer is responsible for the bit-level transmission of data. This service is provided by the transputer link and link adapter. The link is designed to have a failure rate of less than 1 per 10^{25} [13] and for the purposes of the DISTRIX prototype is considered a reliable medium.

Firmware Layer (1)

This layer corresponds to the OSI data link layer and is provided by the transputer at a machine instruction level. Instructions such as the IN instruction blocks on a channel until data arrive and then transfers the data to a memory buffer. A channel may reside on a hardware link

Network Layer (2)

The network layer attaches the necessary addressing information to the message block in the form of a header. The receiver discards the address header, but in the exchange, the header information is used for routing purposes and is passed on intact.

Buffer Layer (3)

This layer is not easily mapped onto any *one* of the OSI layers and its function will be discussed at length later. Essentially it is the highest layer in which specialized message formats are used. Above this layer, standard, localized MINIX messages are used. A characteristic of the buffer layer is that it is always ready to accept messages from the network layer. Thus the network layer is never blocked by unserviced requests.

RPC's, Agents and Counter-Agents

DISTRIX makes use of remote procedure calls[2], and specialized agent processes. Pointers need special attention when passed as parameters in remote procedure calls because of the disjoint address spaces involved. Nelson introduced the concept of *marshalling*, namely the expansion of each pointer reference to include the data to which it points. The reverse process is termed *unmarshalling.*

Two marshalling methods exist at implementation level. With *inline marshalling*, the marshalling code is produced by the compiler, while with *stubs*, library routines perform the marshalling. In DISTRIX, the stub concept has been implemented at the library level for all remote system calls. In order to avoid confusion, the terms *agents* and *counter-agents* (discussed in Section 5) are used to distinguish between the unmarshalling and marshalling sides respectively. The mini-kernel described earlier acts as a marshaller.

5 The File Server

The file server (FS) constitutes more than just the naming of files and providing users with the mechanisms for reading and writing files. It extends to controlling access (according to permission and privilege levels) and providing an organizational structure, such as a hierarchical directory service. It also keeps track of the physical location of a file on the device.

When seen in the context of the UNIX operating system [12], the FS is further responsible for access to all devices in the system. Most UNIX devices are represented and manipulated as files. Examples of DISTRIX devices include terminals, printers and the system clock. The clock is not treated as a file, but is needed for the date stamping of modified files.

The file server described below is *not* a Distributed File System, but does form part of a Distributed Operating System in the sense that several components of the operating system have been separated from one another. By contrast, a Distributed File System consists of a number of file servers managing a group of files cooperatively [6].

Design

One of the aims of this adaptation has been to retain as much of the original MINIX file server as possible. So, rather than introducing many changes to the FS, an environment was created to suit the needs of the FS. This led to some sacrifices, particularly with regard to performance, but the advantages of a near-original FS were evidenced by the relative ease of implementation.

As a server, the FS must be *fair*, handling all requests in some order, so as never to exclude any one client. To meet this requirement and also make the best use of the built-in scheduling features of the transputer, an infrastructure of communications software and agent processes was created. The agents

[2]Nelson [11, Section 2.1.1] defines a remote procedure call as: '*...the synchronous language level transfer of control between programs in disjoint address spaces whose primary communication is a narrow channel.*'

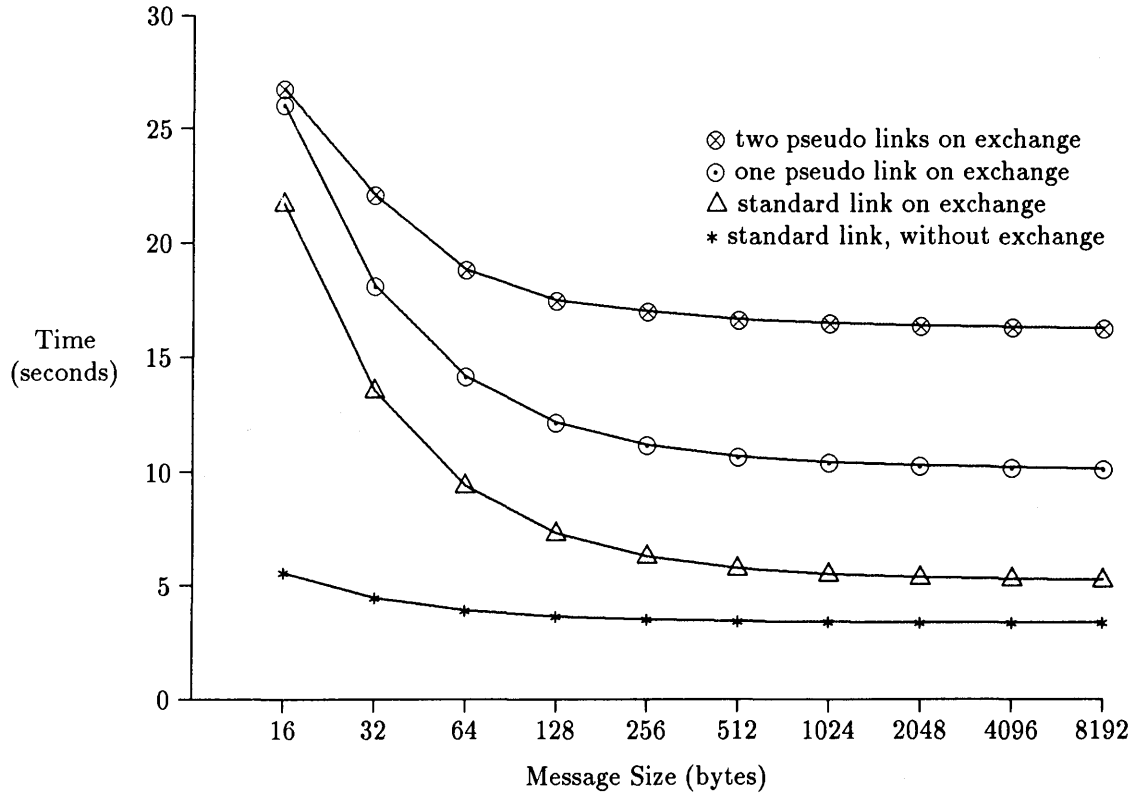

Figure 3: Exchange transfer time as a function of data block size

Exchange Performance

To measure the exchange overhead and assist in tuning the system, the following experiment was performed: One megabyte of data (excluding the headers described later) were transferred in varying packet sizes between two transputers, using the following interconnection:

- a direct connection (omitting the exchange),
- an exchange using two hardware links,
- an exchange using one hardware link and one pseudo link, and
- an exchange using two pseudo links.

The results are shown in Figure 3 while Table 1 shows the exchange throughput at 10Mbit/s for a block size of 1024 bytes. The poor throughput of the pseudo links is caused by software overheads as these links are not serviced by the transputer firmware.

As can be seen from Figure 3, the curves tend to flatten out at a packet size of around 256 bytes. In DISTRIX, the largest block of data ever transmitted is a disk block of 1024 bytes. Counting all headers, the total size is 1050 bytes. As the graph shows a nearly horizontal line for all the curves at that packet size, an efficient transfer is achieved.

Method	Throughput (KB/s)
Theoretical	407
Direct	302
Hard Link	186
Single Pseudo	98
Double Pseudo	62

Table 1: Throughput for transfers of packets of 1024 bytes excluding headers

4 Process Inter-Communication in DistriX

An end-to-end protocol was defined, making use of a layered design after the OSI reference model [17]. Due to the specialized nature of the communications, the model chosen was smaller than the seven-layer OSI model but the design principles were similar.

The goals were to achieve a protocol that was well defined, flexible enough for expansion and efficient within the constraints of the system goals. Consideration also had to be given to the needs of the exchange processor and the remote procedure call mechanism.

A four layer protocol was defined using a message format shown in Figure 4. The protocol supports up to 65536 processors, each with up to 65536 processes.

form part of the remote procedure call mechanism used throughout DISTRIX. The agent processes act not only as marshals, but also as message buffers and, using the scheduler, ensure fair servicing of all work requests.

Several schemes were considered in order to effect controlled buffering of requests arriving at the FS:

- *Explicit Dynamic Buffering*: One process receives all work requests, allocates memory to store each request, and places a request identifier at the end of a queue which the FS services in finite time. The dynamic memory allocation implies that a system call may fail due to memory limitations. This failure may not be reliably predicted by a user.
- *Explicit Static Buffering*: One process receives all work requests, but the buffer slots for every possible client are allocated at system generation time. MINIX already requires that each client has an entry in the process table. This approach solves the unpredictable failure problem noted above, but may limit the number of possible clients allowed in order to accommodate buffer space for each agent process.

In general, the *explicit buffer* approach operates as follows. The buffer layer must accept each incoming work request immediately, so as not to block the network layer. Since the FS runs as a separate process, it has no knowledge of jobs queued, and, in particular, has no knowledge of the order of the jobs. Thus, when the FS has completed its last job, it must indicate its willingness to accept work by means of a message to the buffer layer. The buffer layer, using the queue of requests, chooses the next job and submits this to the FS. This approach requires the buffer layer to choose the next job. Since replies are not buffered, but are written directly to the network layer, this approach displays *asymmetry* with respect to requests and replies.

Hoare [7] describes a buffer as a series of concurrent process, accepting input data from their left and producing output to their right. Following this approach, DISTRIX allocates a process to each buffer.

The DISTRIX variant is called the *implicit buffer* approach and once again has two variations:

- *Implicit Dynamic Buffering*: As new processes are created elsewhere in the system, a corresponding agent process is created simultaneously on the FS. The sole function of this process is to accept work from, and carry out work on behalf of its client sibling on the mini-kernel. The dynamic creation of buffer processes poses the same problem as the explicit buffer approach: memory limitations may cause run-time failure.
- *Implicit Static Buffering*: The same as above, but an agent process is created for every possible client. Each client already has an entry in the process table. The actual process creation takes place at boot-time, but as the buffer space set aside for each process is declared at compile-time, the creation is guaranteed to succeed. This may limit the number of possible clients allowed in order to accommodate buffer space for each agent process.

The choice of which job to run next is thus made by the transputer scheduler and not by the buffer layer[3]. This approach displays *symmetry* with respect to requests and replies, both passing through the agent process (buffer layer).

By means of this buffering, multiple requests for work may be stored simultaneously on the FS processor, but due to its original design, only one request may be carried out at one time. Thus, during a disk fetch (during which time the FS is idle), no other work may be accepted. This contrasts, for example, with a description [15] of the file server in the V Kernel [5]. The V Kernel's FS is described as a team of processes, with one member being able to continue processing whilst another is suspended on some task, such as disk I/O.

Advantages of static buffering

Earlier it was mentioned that dynamic buffering can lead to run-time failure. Whilst it is possible to return an error code when the agent fails to allocate memory for buffering, this would require additional interaction with the central kernel. The central kernel is responsible for administering the process creation. Before allowing the new process to be created, it would thus have to obtain confirmation of the successful creation of the process (or buffer) from the FS.

For the prototype it was decided to eliminate this interaction and create an FS that can guarantee successful buffer creation by performing the allocation at boot-time. As this forms part of a single startup routine, there is no further cost associated with static buffering. The buffer processes are idle until work arrives. On the transputer, idle processes pose no processing overhead.

On the other hand, dynamic buffer creation requires not only kernel interaction, but also memory allocation, even though the time required to allocate memory is small (approximately 1 microsecond).

Implementation

Implicit static buffering has been implemented in the DISTRIX prototype. Each agent process has its own local buffer space and has its own entry in the FS processor's scheduling table. The program code is shared to conserve memory.

When a work request is received, the agent attempts to pass the unmarshalled request to the FS. Should the FS not be ready to service the request (because it is busy with a previous task), the agent is

[3]This was later found to be insufficient. Inherent favouritism exists in the scheduling algorithms and the main program of the FS has to adjust a table to provide 'fair' servicing of all queued requests. This problem is described later in 5.

automatically suspended by the built-in scheduler until such time as the FS becomes ready to receive work.

At this point, one of the descheduled agents is rescheduled and sends the request (along a channel) to the FS in the standard MINIX message format. This use of the MINIX messages meets the design goal of leaving the FS itself largely untouched. Furthermore, the burden of scheduling and queuing has been transferred to the transputer microcode level.

Device Drivers

Logically, all device drivers reside on the FS processor, but the concept of a counter-agent allows for the physical removal of the driver to another processor. When the FS requires the services of such a driver, it becomes a client. To the FS, the counter-agent appears to be the device driver. The counter-agent then communicates with its respective agent process on the device driver processor using the exchange and the associated protocol.

The relocation of device drivers to dedicated processors was tested during early stages of development, but subsequent device drivers made use of high level, intelligent devices. Thus their processing requirements were low and did not require the use of a dedicated processor. Additionally, the protocol overhead and the extra burden placed on the exchange would lead to inefficiencies. Devices attached directly to the FS processor make use of specialized, optimal protocols.

Currently, the disk and terminal devices are emulators of existing or potential hardware equivalents. Both have well defined, high level interfaces. As emulators their performance is poor in comparison with dedicated hardware devices.

Problems

Differences in word size between the Intel and Inmos processors lead to difficulties in placing the FS on the transputer. The transputer does not readily support a 16-bit data type, and the C compiler used does not provide for such a type — shorts, ints and longs are all 32-bit. Several data structures that relied on 16-bit types had to be redesigned.

The C compiler and library provide for delayed execution of processes. One of the criteria for choosing the process is whether a channel has data available. The process wishing to receive data from a channel or collection of channels may be suspended until the data become available [10]. A list of channel addresses must be provided. However, as in Occam, the selection of a channel, if more than one channel has data available, is not fair [9]. The implementation of both Occam alternation and the ProcAltList() function in the C library favour the channels near the head of the list. Channels placed near the tail of the list may be excluded indefinitely. In order to overcome this, the FS changes the order of channels in the list after each job, placing the most recently active channel at the back. By using a double length list and an adjustable end-of-list marker, the time overhead of this operation is negligible.

Performance

Figure 5 shows initial results of some performance measurements that were done on the FS. The indicated system calls were issued by a user process on a satellite processor. The numbers displayed next to the *read* and *write* system calls indicate the number of bytes involved in the transfer. The following times were recorded for each system call:

1. The total time to complete the system call as measured by the user process.
2. The time to complete the system call as measured by the FS only. Note that the difference between the total time in 1 above and this time provides the remote procedure call overhead.
3. The total time to complete the system call as measured by a user process under standard MINIX running on an 8MHz 80286, equipped with a fixed disk.

The initial impression is that a DISTRIX system call takes much longer than one in MINIX (the average for the given figures is 3 times as long). The worst cases are with reading and writing the large (2–4KB) blocks of data. The source of this inefficiency lies in the communication overhead (85%) and further investigations are underway to reduce this. As for the FS itself, an average decrease in execution time of a factor 3 was observed, indicating that the transputer version of the FS is capable of higher performance than the MINIX version, in spite of a relatively slow disk device emulator.

The foregoing measurements were made in a single-process situation. Measurements were also made with the system under load. A background process, designed to consume CPU cycles by performing endless calculations was placed on the user processor. Since the communications overhead in DISTRIX is the major contributing factor to the cost of remote system calls and the additional load placed on the user processor does not affect the exchange or FS processors, the resultant system call execution times are only marginally longer (6.6%). Conversely, in MINIX, the background load process is contending with both the benchmark process and the operating system, slowing both down considerably. MINIX executes approximately 20 times slower than DISTRIX in these circumstances. The graphs are not presented as the factors are so large as to make comparisons meaningless.

Further benchmarks were performed to determine the effects of multiple, simultaneous file server requests. For example, the results for two simultaneous calls are presented in Figure 6. As expected, the times for single processor MINIX (and the DISTRIX file server

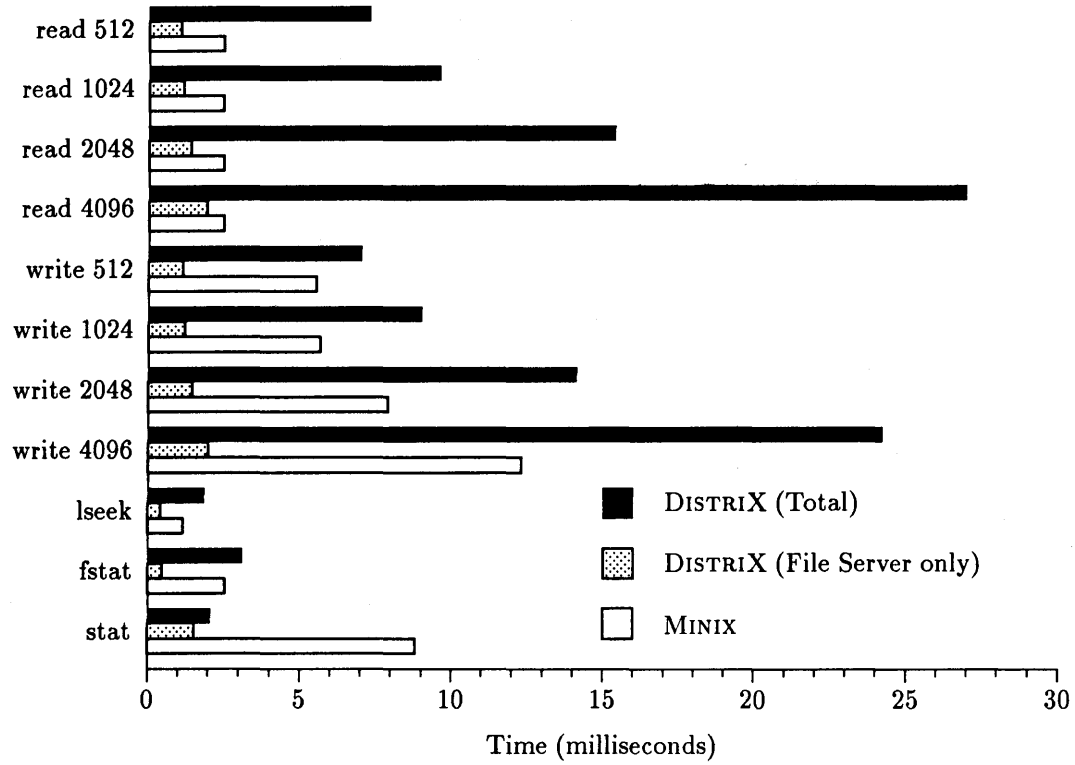

Figure 5: A comparison of the times to perform selected FS system calls

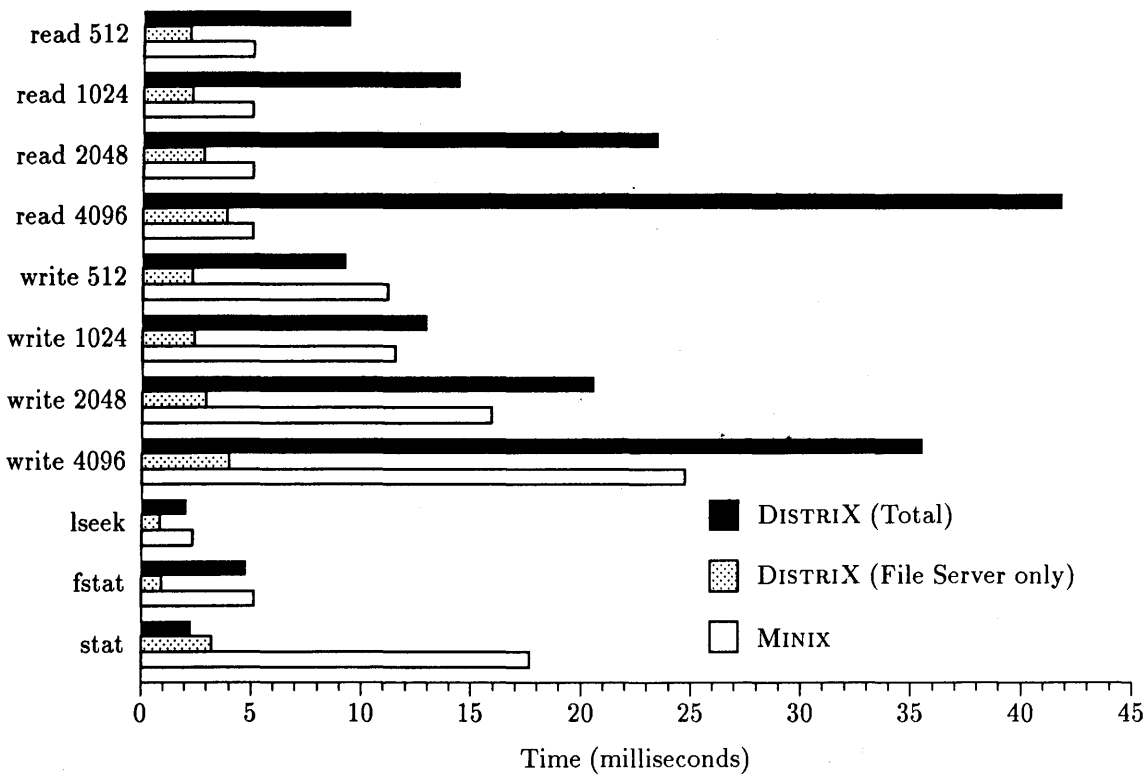

Figure 6: FS performance under double load

Number of processes	Percentage Increase	
	MINIX	DISTRIX
2	100	32
3	200	67
4	300	94
5	400	119

Table 2: Average increase in time for multiple processes

only) increased linearly (100%). The average times measured by the DISTRIX user process increased by only 32%.

The results for up to 5 simultaneous user processes issuing file server requests are summarized in Table 2. It shows the average increase in system call time for MINIX and DISTRIX. The percentages given for MINIX are extrapolated estimations whilst the times for DISTRIX were actually measured. Although the percentage increase for DISTRIX does grow, it remains sublinear for 5 processes. Further measurements could not be made due to memory limitations of the prototype.

The foregoing affirms that the advantages of a multiprocessor system are not realized until the system is placed under load. The mechanisms that provide these advantages in the prototype use considerable resources and provide no return under light load.

6 Conclusions

We have shown that MINIX can be distributed on a network of transputers. The file server can be removed from the main processor and be made to function independently of the central kernel. It can accept work from any process in the system (including the central kernel). It may have devices either attached directly to its own processor, using the transputer links, or attached to other processors (such as the original Intel 80x86 MINIX central kernel processor) using counteragents to send work requests.

The transputer's concurrency, scheduling and communications features have been used to simplify the task of the system level programs, but care had to be taken to avoid scheduling favouritism.

The current design of the FS (based on MINIX) only makes provision for a single file system request to be serviced at a time. Other projects are currently involved in parallelizing the FS to make better use of the concurrency offered by the transputer.

The bottlenecks observed in the system still allow it to operate at an acceptable speed, but improvements in those areas could lead to improvements in I/O performance more in line with the performance improvements noted in CPU bound jobs, where linear speedups have been achieved by increasing the number of processors. As expected, DISTRIX shows a lower performance penalty under load than does an equivalent uniprocessor system.

Attachment of real devices (rather than slow emulators for the disk and terminal interface module) will also show a marked improvement in speed. This can be borne out by a greater than linear slowdown when I/O bound jobs are writing to disk, versus a predicted linear slowdown when reading.

References

[1] Bach, M.J. 1986. *The design of the* UNIX *operating system.* Prentice-Hall, Englewood Cliffs, New Jersey.

[2] Bell. 1983. UNIX *time-sharing system:* UNIX *programmer's Manual.* Holt, Rinehart and Winston, Saunders College Publishing.

[3] Birrell, A.D., and Needham, R.M. 1980. A universal file server. *IEEE Transactions on Software Engineering.* Vol SE-6, 5, September, pp450-453.

[4] Bourne, S.R. 1983. *The* UNIX *system.* Addison Wesley, UK.

[5] Cheriton, D.R. and Zwanepoel, W. 1983. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems principles. Operating Systems Review.* Vol 17, 5, Special Issue, 10-13 October, pp129-140.

[6] Coulouris, G.F., and Dollimore, J.B. 1988. *Distributed systems: Concepts and design.* Addison Wesley, UK.

[7] Hoare, C.A.R. 1985. *Communicating sequential processes.* Prentice-Hall International, UK.

[8] Inmos. 1988. *Transputer reference manual.* Prentice-Hall International, UK.

[9] Jones, G. 1989. Carefully scheduled selection with ALT. *Occam User Group Newsletter.* Inmos Ltd, UK.

[10] Mock, J. 1988. Processes, channels and semaphores (Version 2). *Logical Systems C Compiler User's manual.*

[11] Nelson, B.J. 1981. Remote procedure call. *XEROX PARC Report number CSL-81-9* also published as *Carnegie-Mellon University report CMU-CS-81-119.*

[12] Ritchie, D.M., and Thompson, K. 1978. The UNIX time-sharing system. *The Bell System Technical Journal.* Vol 57, 6, July-August, pp1905-1929.

[13] Shepherd, R. 1987. Extraordinary use of transputer links. *Inmos Technical Note 1, 72-TCH-001-00.* February, Inmos Ltd, UK.

[14] Smit, G. de V., Hoffman, P.K., and McCullagh, P.J. 1989. DISTRIX: A multiprocessor UNIX workbench. *Technical Report number CS-89-04-00.* Department of Computer Science, University of Cape Town, South Africa.

[15] Tanenbaum, A.S., and Van Renesse, R. 1985. Distributed operating systems. *Computing Surveys.* Vol 17, 4, December, pp419-470.

[16] Tanenbaum, A.S. 1987. *Operating systems: Design and implementation.* Prentice-Hall, Englewood Cliffs, New Jersey.

[17] Zimmermann, H. 1980. OSI Reference model – The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications.* Vol COM-28, 4, April, pp425-432.

Acknowledgements

Thanks are due to Paul McCullagh and Gavin Gray for their assistance in design and hunting down some particularly elusive bugs. Pieter Bakkes and Elize van der Walt at the Institute for Electronics at the University of Stellenbosch and Ralph Pina and Gunther Dörgelon at SED at the University of Stellenbosch patiently endured many technical queries and several unscheduled 'adjustments' to our hardware. The Institute for Electronics developed the exchange hardware and the initial exchange software.

ICL(SA) Pty Ltd and Westervoort Research and Development provided financial assistance and equipment.