An Enhanced Transputer Server Architecture

Brendan Murphy
Department of Computer Science, University of Buckingham, MK18 1EG, UK
brendan@buck.ac.uk

Abstract. This paper describes an Enhanced Transputer Server Architecture (ETSA). The purpose of ETSA is to provide a more flexible and sophisticated environment for interacting with transputer systems than is provided by existing server models. ETSA is a network-based architecture that permits the user of a transputer system to be remote from that system. ETSA supports arbitrary server communication protocols such as the iserver (SP) protocol and the af-server (AFS) protocol. This means that existing transputer software can run over ETSA without change. An additional ETSA server has been implemented that provides a number of enhancements to the Inmos iserver (when supported by the underlying operating system). These enhancements include multi-threading, support for dynamic server extensions, and support for multiple X window pseudo-terminals.

1. Introduction

The Inmos transputer is an ideal component for building parallel systems. A typical transputer comprises a CPU, four DMA-driven serial communication links, high resolution timers, on-chip memory and a floating-point unit. Although a transputer integrates much of the functionality of a computer on a single integrated circuit, additional peripheral devices are normally required to facilitate the storage and retrieval of data.

A general-purpose computer has a disk filing system and terminal/network support in order to provide access to the CPU from the outside world. Transputers have been used to build such standalone systems hosting operating systems such as Helios. Frequently, however, transputer systems are configured as slaves for popular hosts such as the IBM PC and Sun workstation. Such systems are accessed via plug-in interface boards that usually map part of the host i/o address space to a transputer link adapter. Rather than having an independent i/o subsystem, these *slave* transputer systems make use of the i/o subsystem of the host computer. It is this class of transputer system that is addressed in this paper (Figure 1).

In order to make the host's i/o subsystem accessible to the target transputer system, the host runs a program called a "server". Typically, the server is also used to boot the target (slave) system at start of day. The server reads incoming requests from the transputer target (via the interface board), processes the request, and sends an appropriate response back to the target. The format of such requests is dictated by the server communication protocol. The server is often written in a high level language such as C and is designed to be portable to a number of host platforms.

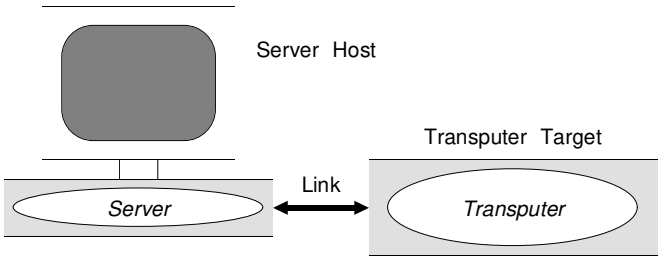

Figure 1: The Conventional Transputer Server System

This paper identifies a number of limitations of conventional transputer servers. It then presents the Enhanced Transputer Server Architecture and discusses the benefits provided by this architecture.

Two implementations of ETSA have been performed. The first implementation is layered above the ANSAware distributed computing infrastructure which is compliant with the Advanced Network Systems Architecture (ANSA) [1]. ANSAware provides a platform-independent multi-threaded environment for writing distributed systems. The second implementation is layered above the Berkeley Unix BSD socket interface and the TCP/IP protocol suite [2]. The BSD socket interface is a de-facto standard for writing distributed applications; it has been ported to a wide variety of platforms. This paper describes the salient features and relative merits of the two implementations.

A set of performance measurements is presented to compare the efficiency of the two implementations under a variety of operating conditions. The results are also used to compare the performance of ETSA with a conventional (non-ETSA) server model. A number of changes to the transputer i/o libraries are proposed to improve the performance of ETSA and other network-based server systems in which network latency is a significant factor in overall throughput.

2. Limitations of Standard Servers

- Many users work in a LAN environment in which the ability to access remote resources such as printers and fileservers is taken for granted. Conventional transputer server programs do not have the capability of accessing transputer resources that are hosted by remote machines. *This limits the scope for resource sharing.*
- The i/o primitives provided by standard server programs such as `iserver` and `af-server` reflect the base set of system and library calls that are provided by conventional operating systems. Transputer applications that demand a richer set of primitives require a custom-built server to support these primitives. *This involves reprogramming the server on a per-application basis.*
- Standard server programs deal with i/o requests synchronously: an incoming request has to be processed and a reply generated before the next request can be serviced. A request that causes the server to block (eg waiting for a key to be pressed on the keyboard) will stall all other client transputer processes that require access to the server. *This represents a serious i/o bottleneck, especially in the context of a multiprocessor target.*
- An additional problem with standard servers is that the transputer application has access to only one terminal, namely the terminal associated with the server itself. Terminal i/o requests from multiple transputer processes are multiplexed onto a single terminal. Most host operating systems support windowing systems that allow multiple pseudo-terminal windows to share the resources of the (single) physical terminal. *Standard server programs do not provide access to such windows.*

3. Motivation for Designing ETSA

The motivation for designing the Enhanced Transputer Server Architecture (ETSA) was to provide a more flexible and sophisticated environment for interacting with transputer systems than is provided by existing server models. In the light of the limitations outlined above, four main requirements were identified:

- The need to make the transputer system accessible to remote hosts when the transputer host is part of a networked system (eg one of several machines on a LAN).
- The need to provide multiple "threads" of execution within the server each of which can proceed in parallel.
- The need to extend dynamically the functionality of the server in an application-specific manner without having to rewrite the server.
- The need to allow multiple transputer processes to perform terminal i/o in separate server windows rather than having terminal i/o from multiple processes multiplexed onto a single terminal.

Network access to transputer systems has been provided by other systems such as PCserver [3] and the Inmos B300 [4]. Multiple pseudo-terminals and dynamic server extensions have been provided by systems such as WIServer [5] for Microsoft Windows. ETSA offers most of the features provided by these systems with the addition of a multi-threaded server. Furthermore, ETSA is portable to a wide variety of platforms and is extremely flexible in its configuration.

4. The Enhanced Transputer Server Architecture

4.1. Structure

The structure of an ETSA system is shown in Figure 2.

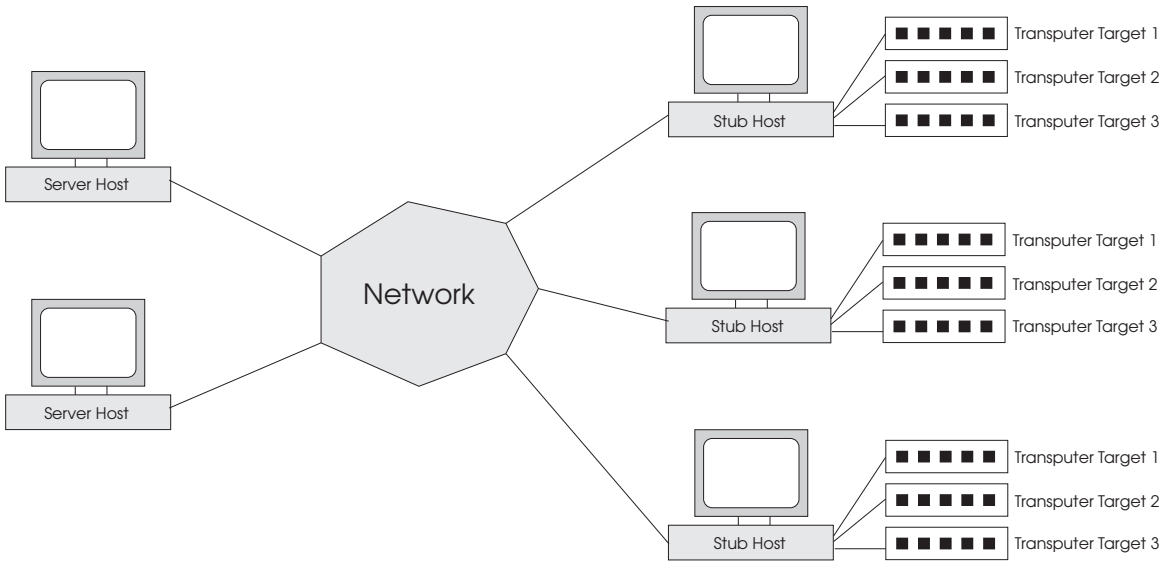

Figure 2: ETSA Structure

The key feature of ETSA is that the code that handles the transputer interface is in a different module to the code that interacts with the user. This allows the two modules to be separated by a network. From an architectural perspective, the situation in which the two modules are co-located is identical to the more general case in which the two modules are separated. From an engineering perspective, performance optimisations are possible in the local case.

The module that handles the transputer interface is termed the "stub". The module that interacts with the user is termed the "server". The stub transparently relays requests and replies between the transputer and the server.

The stub runs on a processor with access to a network interface and one or more transputer interfaces. The stub may be hosted on a network TRAM but is more likely to be hosted on a general-purpose machine equipped with a network interface such as an IBM PC or a Sun workstation. The stub can handle *multiple* target transputer systems each of which is available to a separate user via the network.

A server gains access to a transputer system by binding to an appropriate stub and selecting an appropriate target. Thereafter, the server is able to boot the target before starting to handle i/o requests from the target.

The server code that interacts with the stub is contained in a single ETSA link module; existing servers (such as `iserver` and `afserver`) that wish to use ETSA simply link the ETSA link module in place of their standard link module. This gives existing servers the benefit of distribution; that is, the server may communicate with any other transputer resource on the network. However, in order to take advantage of the additional benefits of ETSA, a new `iserver`-compatible server has been written. The ETSA server supports multi-threading, dynamic server extensions, and multiple X window pseudo-terminals.

4.2. Multi-threading

Each request from a client transputer process is handled by a separate server thread. If a request cannot be serviced immediately (typically because it is a blocking i/o request such as a read from the keyboard), only the thread handling the request is blocked, *not the entire server.* Furthermore, if the i/o operation cannot be completed in one non-blocking system call (as might be the case with a slow peripheral), the thread repeatedly suspends itself between system calls until the operation is complete. The multi-threaded implementation of the ETSA server permits a greater degree of concurrency than is possible with a single-threaded server. In particular, the use of multiple threads removes the synchronous i/o bottleneck that is a limitation of conventional transputer servers.

Conventional server programs employ a single-threaded request-reply link communication protocol. Accordingly, the transputer i/o library procedures that communicate with the server perform synchronous i/o: the procedure sends a request and blocks until it receives a reply. The CSP communication convention is that a channel is used for communication in only one direction and between only two processes [6]. Since the server is accessed via a pair of (uni-directional) channels (mapped onto a physical link), only one process can communicate with the server. In an application in which multiple processes require access to the server, a multiplexor process is required to field requests on behalf of the clients. If one of the clients makes a blocking request to the server, the multiplexor process will block access to the server from all other clients. This communication paradigm is consistent with the single-threaded nature of conventional server programs.

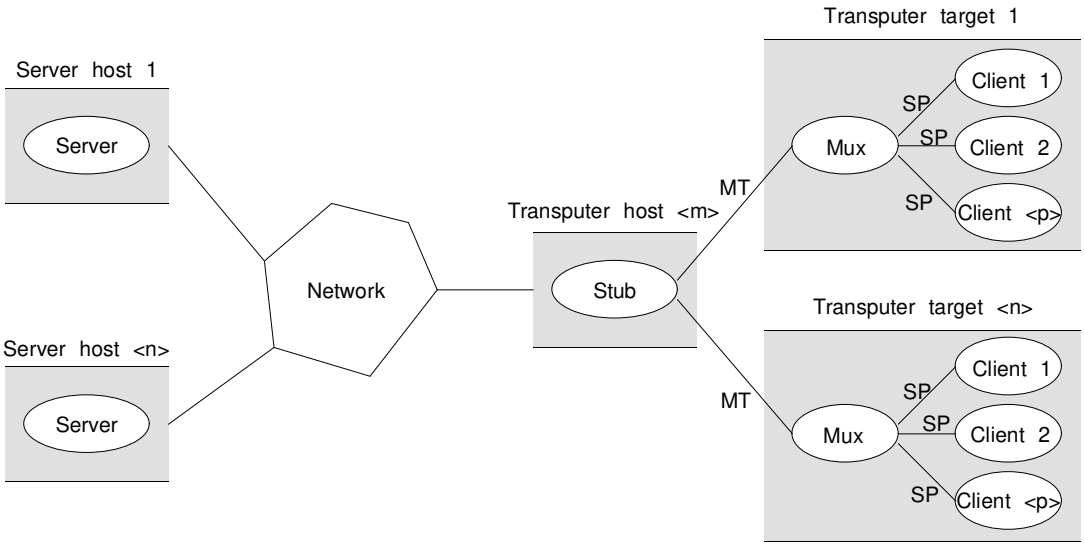

Figure 3: ETSA Multiplexing Architecture

In order to exploit the multi-threaded functionality of the ETSA server, the single-threaded link protocol needs to be replaced by a multi-threaded link protocol in which request packets are tagged to identify their owner. Furthermore, the synchronous multiplexor process needs to be replaced by an asynchronous multiplexor process which does not block waiting for replies. Instead, the multiplexor inserts a tag into the request packet in order to identify the owner before transmitting it to the server; this tag is returned in the reply packet and used to demultiplex reply packets to the appropriate client. The asynchronous multiplexor uses the standard link protocol on the client side but the multi-threaded link protocol on the server side. This communication paradigm allows multiple server transactions to be in progress simultaneously. If one server transaction is blocked, other transactions can still proceed.

The ETSA server uses the MT (multi-threaded) link protocol in place of the standard `iserver` SP link protocol. The MT protocol is a simple extension to the Inmos SP protocol: every packet includes an additional byte (called the client identifier) that identifies the owner of the packet. It is important to realise that the MT protocol is only used on the link between the MT multiplexor and the stub; the SP protocol is used between client and MT multiplexor thereby obviating the need to change the client libraries. Multiple SP protocol sessions are carried over a single MT protocol session. Each MT protocol session is handled by a single stub-server session. This arrangement is shown in Figure 3.

By default, the ETSA server uses the SP link protocol which makes the server upwardly compatible with the Inmos `iserver`. A command-line switch is required to instruct the ETSA server to use the MT link protocol. This means that transputer applications can utilise the other server enhancements without needing to run the MT protocol.

4.3. Dynamic Server Extensions

One of the design goals identified in Section 3 was that ETSA should support dynamic extensions to server functionality on a per-application basis. It was decided that the ETSA server would adopt the Unix `fork()/exec()` model to realise this objective on the basis that such a model is supported by a large number of multi-tasking operating

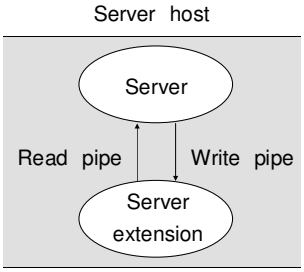

Figure 4: Server Extensions

systems. The steps taken by the ETSA server when spawning a server extension are outlined below:

- The server creates a pair of (uni-directional) inter-process pipes for communication with the child process.
- The server forks a child process which will be used to run the program specified by the transputer client.
- The child process plugs the end of one pipe into its standard input and the end of the other pipe into its standard output. It then closes all other file descriptors that have been passed to it by its parent.
- The child process constructs an argument vector to be passed to `exec()` based on parameters supplied by the transputer client.
- The child process "execs" the specified program.
- The server passes back to the client a file descriptor that references the write pipe. The server maps future requests to read from this file descriptor onto requests to read from the read pipe. Likewise, the server maps future requests to close this file descriptor onto requests to terminate the process and close both pipes.

This arrangement is shown in Figure 4.

In order to spawn a server extension process, a client uses the standard `open()` library procedure supplying a special name of the form "`SYSTEM:<SystemString>`". The server traps such file open requests and interprets `<SystemString>` to be the name of the server extension program to run together with optional program arguments. The server can spawn multiple server extension processes.

From the client's point of view, the `SYSTEM` process looks like a standard file with a corresponding file descriptor. Reading from the file descriptor causes data to be read from the standard output of the `SYSTEM` process. Writing to the file descriptor causes data to be written to the standard input of the `SYSTEM` process. Closing the file descriptor causes the `SYSTEM` process to be terminated.

The server extension program has access to all the facilities of the host operating system. For example, the program may be a graphical front-end to a transputer application or network daemon offering a transputer-based service to TCP/IP clients. The important point to understand is that no special libraries are required by the transputer application; the application communicates with the `SYSTEM` process using the conventional `read()/write()` client library procedures. The ETSA philosophy on server extensions is that it is easier to provide a standard mechanism by which a transputer application can make use of the services of the host operating system rather than to port those services directly to the transputer.

4.4. Multiple Pseudo-Terminals

Another design goal identified in Section 3 was that ETSA should support multiple pseudo-terminals. It was decided that the ETSA server would use X Windows to realise this objective on the basis that X is a freely available windowing system that has been ported to a wide variety of platforms. Furthermore, an X11 terminal emulator program (`xterm`) already exists that can be used in "slave" mode as a terminal for other applications. The use of `xterm` in this way also relies on the Unix `fork()/exec()` model mentioned above. The steps taken by the ETSA server when opening a pseudo-terminal are outlined below:

- The server opens a pseudo-terminal (pty) which comprises master and slave devices.

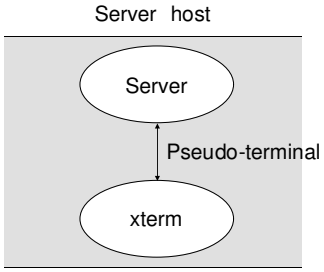

Figure 5: Multiple Pseudo-Terminals

- The server forks a child process which will be used to run the `xterm` program in slave mode.
- The child process closes all file descriptors that have been passed to it by its parent except the master pty device.
- The child process constructs an argument vector to be passed to `exec()` based on parameters supplied by the transputer client.
- The child process "execs" the `xterm` program in slave mode passing it the file descriptor of the master pty device and any additional arguments specified by the client.
- The server passes back to the client a file descriptor that references the slave pty device.

This arrangement is shown in Figure 5.

In order to open a private terminal emulator window, a client uses the standard `open()` library procedure supplying a special name of the form "`XTERM:[<args>]`". The server traps such file open requests and interprets `<args>` to be optional arguments to the `xterm` program.

From the transputer client's point of view, the pseudo-terminal looks like a standard file with a corresponding file descriptor. Reading from the file descriptor causes data to be read from the keyboard if the window has the keyboard focus. Writing to the file descriptor causes data to appear in the `xterm` window. Closing the file descriptor causes the `xterm` process to be terminated.

Two ETSA message types need to be added to the Inmos SP protocol in order to provide full support for multiple pseudo-terminal windows:

- The `ETSA_READCH` message is needed to support non-echoed reading of a character from a specified pseudo-terminal. This is analogous to the `SP_GETKEY` message which reads from the server's standard input.
- The `ETSA_POLLCH` message is needed to support polling for terminal input from a specified pseudo-terminal. This is analogous to the `SP_POLLKEY` message which polls for input from the server's standard input.

These two additional message types require a small number of supplementary library procedures.

5. Implementation

The first ETSA implementation is layered above the ANSAware distributed computing infrastructure. The second ETSA implementation is layered above the BSD socket interface and the TCP/IP protocol suite [2].

In the ANSAware implementation, an `iserver`-compatible server, `XTserver`, has been tested with the `XTstub` stub program. In the BSD sockets implementation, the standard Inmos `iserver` and an `iserver`-compatible server, `eserver`, have been tested with the `estub` stub program. The `afserver` has not been tested in either implementation, although it should be portable to the BSD sockets implementation with minimum effort.

The ANSAware implementation has been tested on IBM PCs running MS-DOS and Sun3/Sun4 workstations running SunOS 4.1. The BSD socket implementation has been ported to IBM PCs running MS-DOS (using either FTP PC/TCP or Sun PC-NFS TCP/IP

libraries), IBM PCs running Unix System V Release 4 and Sun3/Sun4 workstations running SunOS 4.1. In both implementations, the stub program has only been tested on IBM PCs (running either MS-DOS or Unix System V) due to the lack of interface hardware for other machines.

5.1. The ANSAware Implementation

ANSAware adds value to the underlying operating system by providing a generic set of services across a variety of platforms including PC/MS-DOS, DEC/Ultrix, Sun/SunOS, etc (Figure 6). In the context of ETSA, ANSAware provides the mechanisms for location-transparent binding and access-transparent communication between stub and server.

ANSAware chooses the communication mechanism at run-time depending on whether the stub and server are separated by a network (in which case RPC is used over a variety of possible transport protocols) or co-located within the same host (in which case IPC is chosen as a local optimisation). The access mechanism is transparent to the application.

In ANSAware, each component of a distributed application implements a set of operations defined by an interface. A component wishing to act as server for one of its interfaces does so by registering that interface with the Trader together with a set of properties associated with that interface. A client wishing to invoke an operation on such an interface must first import a reference to that interface by means of the Trader. The Trader matches the set of properties requested by the client with the set of properties offered by the server. The binding mechanism is location-transparent in the sense that the client need not be aware of the location of the server.

In the ANSA implementation of ETSA, the stub registers a set of interface instances with the Trader, one for each of the transputer resources that are being served by the stub. Each interface instance has an associated set of properties that characterises the transputer resource (eg `Transputer=T800`, `Memory=4Mb`). The ETSA server (the ANSA client) binds to an appropriate stub (the ANSA server) using the prop-

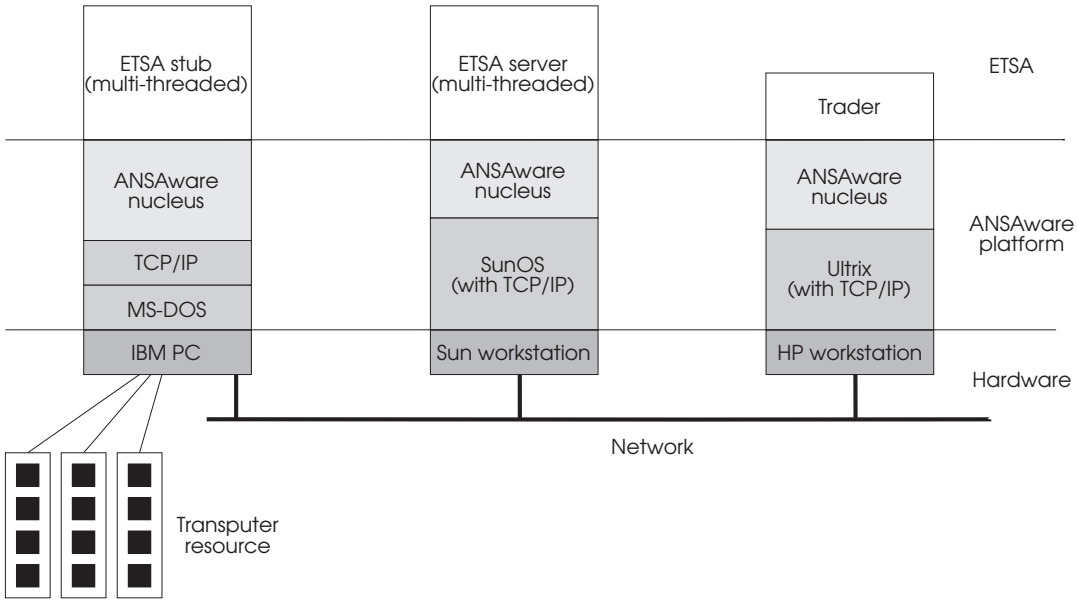

Figure 6: The ANSAware Implementation

erty-matching algorithms of the Trader. The stub interface supports a set of operations that allow the server to interact with the transputer (for example `ResetTransputer`, `BootTransputer`, `PeekTransputer`). Likewise, the server interface (a reference to which is passed to the stub at bind time) supports an operation (`HandlePacket`) that allows the transputer to interact with the server.

The stub and server are inherently multi-threaded due to the run-time environment provided by the ANSAware nucleus. The main thread of the stub polls each of the targets using a simple round-robbin scheduling policy. When a request packet is read from the link, a new thread is created to read the packet from the target, to invoke the `HandlePacket` operation on the server interface, and to write back the result to the target. The stub can handle multiple requests from any of its targets concurrently.

On the server side, a new thread is created to handle each `HandlePacket` operation. Simultaneous access to the same file descriptor is prevented by use of the ANSAware Event Count and Sequencers (ECS) mechanism. A thread guarantees not to make a blocking read system call by registering a PIN[1] handler for the file descriptor in question with the ANSAware thread scheduler and then suspending itself on an event count. The PIN handler is invoked by the scheduler when the file descriptor is ready for reading. The PIN handler simply increments the event count on which its parent thread is waiting. The parent thread can then perform the read operation in the knowledge that the operation will not block.

Furthermore, file descriptors are marked as non-blocking. If the file descriptor references a slow device that cannot completely satisfy the read request in a single non-blocking system call, the thread will repeatedly re-register the PIN handler, suspend itself, and later retry the system call until the read request has been satisfied. This feature maximises the concurrency of the server by preventing a slow peripheral from delaying threads that are reading from other devices.

5.2. The BSD Sockets Implementation

In the BSD sockets implementation, communication between stub and server is achieved by layering an ETSA communication protocol directly over the TCP/IP protocol stack. This arrangement is shown in Figure 7. The format of an ETSA packet is shown in Figure 8. The ETSA programs use the Berkeley 4.3BSD sockets application programming interface; a socket is an abstraction of a network connection that makes it look as much as possible like a conventional file. A socket can be manipulated by standard system calls such as `read()` and `write()`. The sockets interface has been ported to a wide variety of platforms.

Special care was required to preserve the packet semantics of server communication protocols such as the SP protocol over a TCP/IP byte-stream connection. This is because TCP/IP does not guarantee to preserve message boundaries: multiple writes at the transmit end may be matched by a single read at the receive end. ETSA packet boundaries are preserved by exploiting the fact that the first two bytes of the packet represent the length of the packet. The stub and server use special i/o procedures for reading from and writing to the TCP/IP socket that guarantee (except in the case of errors) to transfer exactly one packet's worth of data at a time.

---

1 An ANSAware "PIN" is an abstraction of an operating system file descriptor.

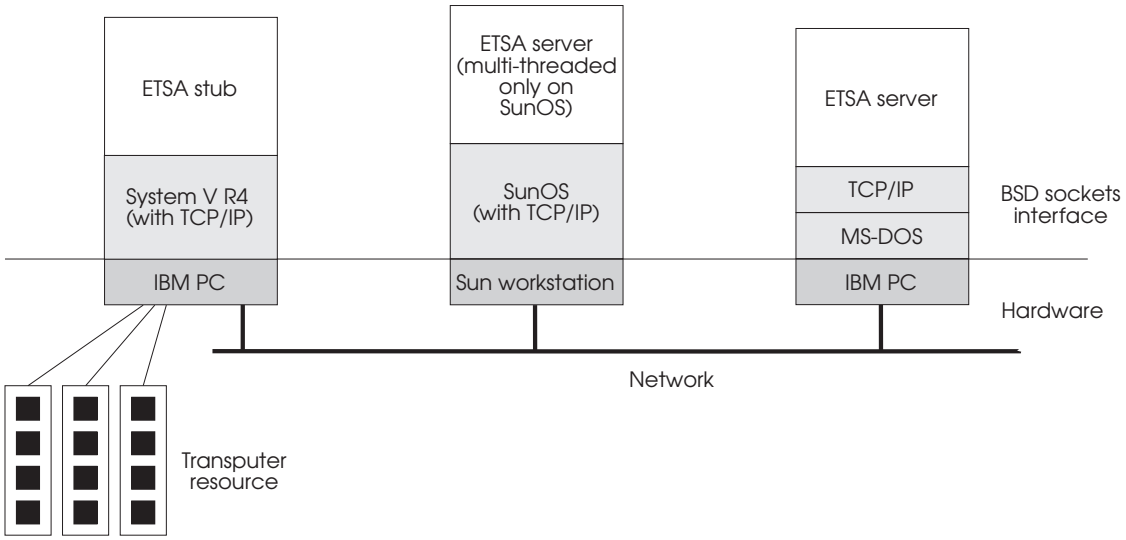

Figure 7: The BSD Sockets Implementation

The ETSA packet types reflect the operations on the stub and server interfaces used in the ANSAware implementation. For example, packet types include `ETSA_RESET`, `ETSA_BOOT`, `ETSA_PEEK` and `ETSA_DATA`.

In lieu of the location-transparent binding facility provided by the ANSAware Trader, the BSD sockets implementation uses an auxiliary program called `ewho` that broadcasts a UDP status request to all stubs on a network and prints the replies. The user chooses an appropriate stub and target on the basis of this information; the property-matching process is not automated.

In the BSD sockets implementation, the stub is a single-threaded entity. The stub marks the public socket on which it receives incoming TCP/IP connection requests as non-blocking. The stub marks all additional sockets representing TCP/IP connections to remote servers as non-blocking. The core of the stub is a tight loop[2] that polls for input from each of its targets, polls for input from each of its servers, and polls for incoming connection requests from new servers. The stub uses a simple round-robbin scheduling policy to ensure that no one target or server gets more than its fair share of processor time. Whenever a request packet is received from a target, the stub sends the packet to the corresponding server before resuming its polling loop. Whenever a reply packet is received from a server, the stub sends the packet to the corresponding

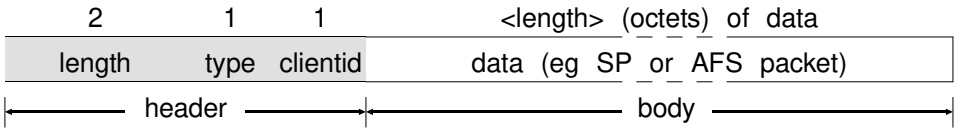

Figure 8: The ETSA Packet Format used in the BSD Sockets Implementation

---

2 The Unix SVR4 version of the stub has a mode of operation that prevents it from hogging the processor: after a short period of link inactivity (several minutes), the stub sleeps each time around the polling loop. The basic problem is that the B004 interface does not use interrupts.

target before resuming its polling loop. Control packets and connection set-up requests are trapped and processed accordingly.

When the server is running on a Sun workstation, the SunOS 4.1 Lightweight Process (LWP) library is used to provide a multi-threaded environment. The LWP implementation of the ETSA server uses two thread-priority levels and relies on the pre-emptive nature of the LWP scheduler. The core of the server is a thread which runs at the lower priority and acts as a custom scheduler. A second thread runs at the higher priority and listens for packets from the network. Additional high priority threads are created by the network listener thread to process each incoming request packet. Whenever a high priority thread needs to read from either the socket representing the server's TCP/IP connection to the stub or a local file descriptor, it sends a message to the scheduler and suspends itself waiting for a reply. Assuming that no other high priority threads are runnable, the scheduler runs and blocks the entire process on a `select()` system call waiting for input from the socket or any local file descriptors. When data is ready on a given file descriptor, the scheduler sends a reply message to the waiting (high priority) thread causing the scheduler to be pre-empted by that thread. The high priority thread either runs to completion or until it next suspends itself by sending a message to the scheduler. The server allows multiple threads to be queued on the same file descriptor, in which case they are handled on a first-in-first-out basis. The use of the custom scheduler thread in this way ensures that a blocking i/o request made by one thread does not block the entire process[3].

Furthermore, all file descriptors are marked as non-blocking. If the file descriptor represents a slow device that cannot completely satisfy an i/o request in a single non-blocking system call, the thread will repeatedly suspend itself on the scheduler and later retry the system call until the i/o request has been satisfied. This feature maximises the concurrency of the server by preventing a slow peripheral from delaying threads that are performing i/o on other devices.

6. Performance

Table 1 summarises the platforms on which ETSA has been tested. The stub programs have been tested on Intel x86 hardware with B004-compatible transputer interface boards

Table 1: Summary of Tested ETSA Platforms

	PC/MS-DOS	PC/SVR4	Sun3/SunOS4.1	Sun4/SunOS4.1
ANSA stub	Yes	No	No	No
ANSA server	Yes[2]	No	Yes[3]	Yes[3]
BSD stub	Yes	Yes	No	No
BSD server	Yes	Yes[1]	Yes[3]	Yes[3]

1 = Server extensions
2 = Multi-threaded
3 = Server extensions and multi-threaded

3 There is no kernel support for multiple threads in SunOS 4.1. In SunOS 5, the kernel is inherently multi-threaded and the LWP library is redundant.

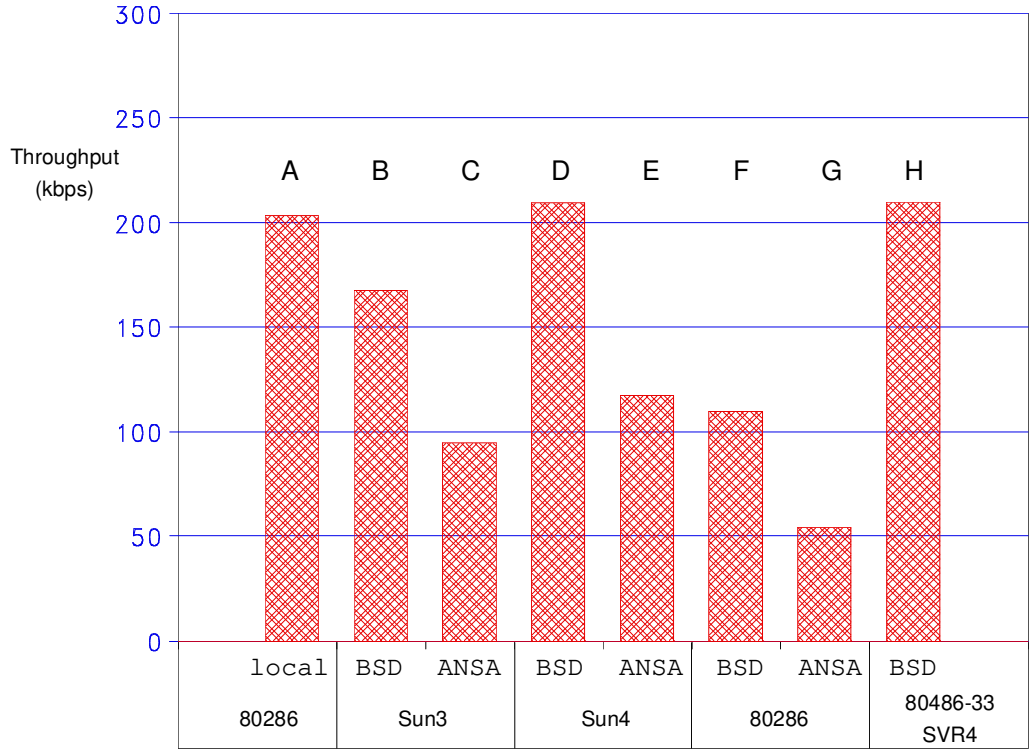

Figure 9: Performance of ETSA using Standard i/o Libraries

running both MS-DOS and Unix SVR4. The server programs have been tested on Sun3 and Sun4 hardware running SunOS 4.1 and Intel x86 hardware running both MS-DOS and Unix SVR4 (except for the ANSA server).

Figure 9 shows the result of running a transputer test program (written in occam using the D7205 toolset) to read 32,000 bytes from a file in units of 500 bytes using the standard (synchronous) Inmos i/o libraries. Bar A shows the speed of transfer using a non-ETSA `iserver` running on an 80286/MS-DOS3.3 PC hosting a T8 transputer. The remaining bars show the effect of using ETSA, the stub running on the same 80286 PC and the server running on four remote platforms: Sun3/SunOS4.1, Sun4/SunOS4.1, 80286/MS-DOS3.3, 80486-33/SVR4.0. The exact measurements are not significant since there are so many parameters beyond the control of ETSA such as disk speed, i/o bus bandwidth, caching policy, network traffic, and network interface performance. However, three general points emerge:

- The ANSAware implementation of ETSA is significantly slower (between approximately 60% and 100%) than the BSD sockets implementation. This is not surprising since the additional functionality provided by ANSAware (particularly the RPC processing) costs CPU cycles.
- The faster the server host, the better the overall performance.
- For fast server hosts (Sun4, 80486-33) and a slow transputer host (80286), the performance of ETSA (over the network) is as good as that of a local `iserver`. Under these circumstances, the faster hardware of the server host compensates for network latency.

Figure 10 shows the result of running the same test program with modified (asynchronous) i/o libraries. As before, bar A shows the speed of transfer using a non-ETSA

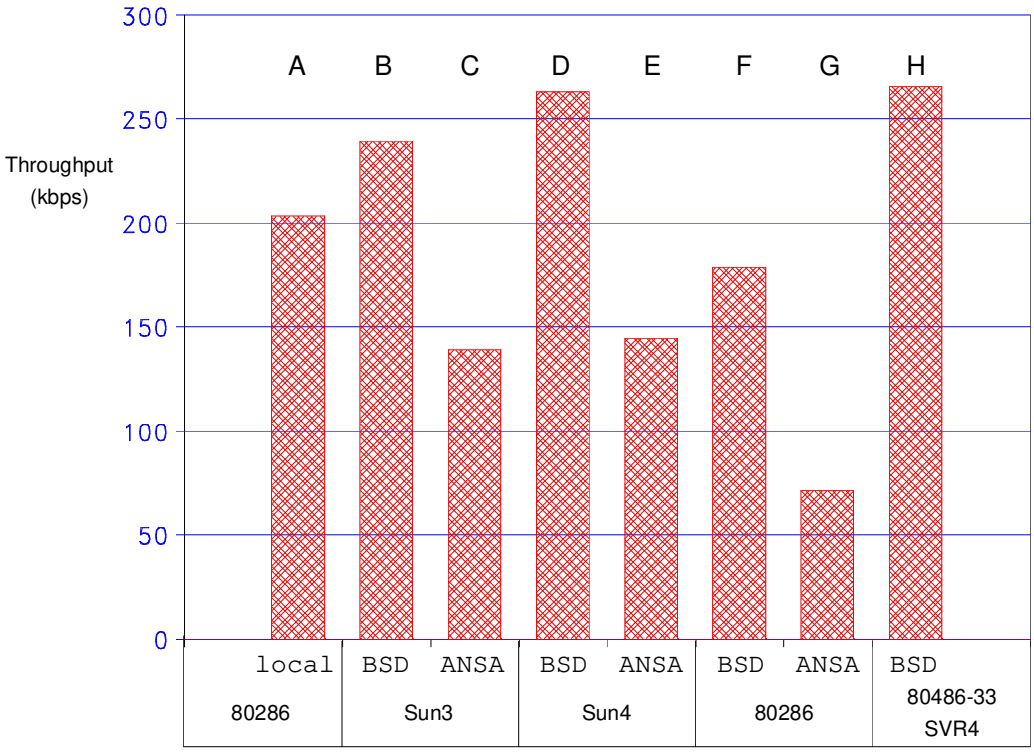

Figure 10: Performance of ETSA using Custom i/o Libraries

`iserver` (using standard i/o libraries) running on an 80286/MS-DOS3.3 PC hosting a T8 transputer. The libraries work by segmenting a client i/o request into a series of smaller i/o requests and sending all requests to the server *before* waiting for replies. As the reply packets arrive, they are re-assembled to generate the single reply packet that is returned to the client. This is possible because ETSA guarantees to maintain packet sequentiality. A buffer is required between the libraries and the stub to prevent deadlock on the link. The advantage of this approach is that it eliminates the round-trip delay that is otherwise incurred for each i/o transaction thereby improving server utilisation. This optimisation provides a significant improvement in performance for situations in which transputer-server communication latency is high relative to packet processing time (for example, when the transputer and server are separated by a relatively slow network such as Ethernet). Specifically, throughput in this test is improved by between 20% and 30% compared with the measurements depicted in Figure 9.

In reality, the above test scenario does not provide a good measure of performance. In practice, data is transferred between transputer and server in variable length packets. Small packets suffer a relatively high overhead in ETSA, as in most network systems. A functional test using production software is necessary to provide a fairer indication of performance. Figure 11 shows the times taken to compile a small occam program (`simple.occ`) on a variety of platforms using a transputer-hosted toolset (INMOS D7205). Bar A shows the compilation time using a non-ETSA `iserver` running on an 80386/MS-DOS5.0 PC hosting a T8 transputer. The remaining bars show the effect of using ETSA, the stub running on the same 80386 PC and the BSD sockets version of the server running on four remote platforms: Sun3/SunOS4.1, Sun4/SunOS4.1, 80386/MS-DOS5.0, 80486-33/SVR4.0. Compilation times are between 10% slower (on

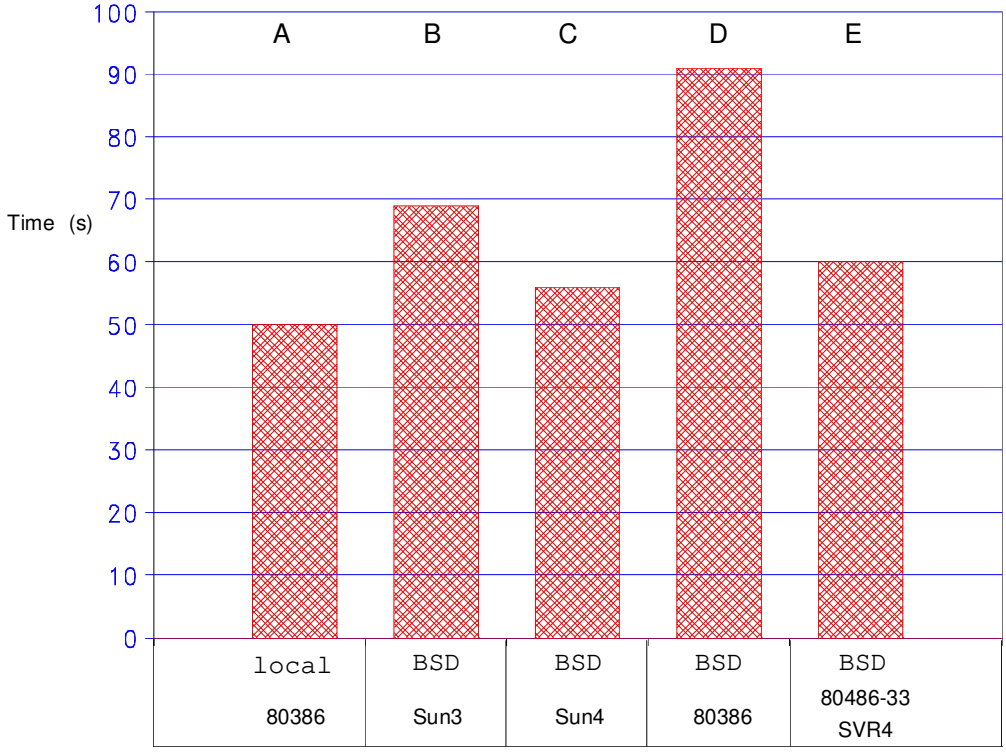

Figure 11: Compilation Times on Various Platforms

the Sun4) and 80% slower (on the 80386) using ETSA compared with using a local `iserver`. The discrepancy would be smaller for a slower (80286) transputer host machine with less filing system cache (640KB of RAM on the 80286 compared with 4MB of RAM on the 80386). The discrepancy would be further reduced if the compiler used the asynchronous i/o libraries outlined above in place of the standard i/o libraries. A further opportunity for improvement is simply to increase the maximum size of the buffers used by the standard i/o libraries from 512 bytes to 1024 bytes, the maximum packet size supported by the SP link protocol.

Two more results are worthy of mention. Firstly, the Unix System V Release 4.0 version of the BSD sockets stub program is approximately 15% slower than the MS-DOS version of the same program using the compilation test mentioned above. This is because the multi-tasking nature of Unix imposes a greater overhead than the single-threaded nature of MS-DOS. The attraction of running the stub on Unix, however, is that the machine can still be used for other purposes. Secondly, although the processor can multi-task the MS-DOS stub with other application programs under MS Windows, performance is approximately 150% worse (not shown in the Figure) under MS Windows compared with MS-DOS (even when no other Windows processes are active). This is due to the simplistic time-slicing policy of the MS Windows scheduler.

7. Conclusions

ETSA is a network-based architecture that permits the user of a transputer system to be remote from that system. The distributed nature of ETSA improves the ac-

cessibility of possibly scarce or expensive transputer resources. Furthermore, access is possible from a variety of host platforms such as Sun workstations running SunOS/Solaris, DEC workstations running Ultrix, IBM PCs running MS-DOS/SVR4, etc. Economic benefits are achievable by hosting transputer resources on platforms for which low-cost interface hardware is available; ETSA allows these resources to be accessible to other hosts on the network. In particular, ETSA allows *multiple* transputer systems hosted by a single low-cost IBM PC to be accessible to multiple remote users.

ETSA supports arbitrary server communication protocols such as the `iserver` (SP) protocol and the `afserver` (AFS) protocol. This means that existing transputer software can run over ETSA without change. An additional ETSA server has been implemented that provides a number of enhancements to the Inmos `iserver` (when supported by the underlying operating system). The ETSA server enhancements and their key benefits are summarised below:

- **Multi-threading**. Each request from a client transputer process is handled by a separate server thread. If a request cannot be serviced immediately (typically because it is a blocking i/o request such as a terminal read), only the thread handling the request is blocked, *not the entire server.* The multi-threaded implementation of the ETSA server permits a greater degree of concurrency than is possible with a single-threaded server. In particular, the use of multiple threads removes the synchronous i/o bottleneck that is a limitation of conventional transputer servers.
- **Dynamic server extensions**. The server allows a client transputer process to spawn arbitrary host processes, the standard output and standard input of which are piped to and from the client. This facilitates dynamic extensions to server functionality on a per-application basis without requiring customisation of the server itself.
- **Multiple X window pseudo-terminals**. The server allows a client transputer process to open one or more X window pseudo-terminals thereby permitting multiple transputer clients to share the resources of a single physical terminal. This is particularly useful when a transputer application comprises multiple processes each of which requires access to a separate terminal (for example, for the display of debugging information).

Two implementations of ETSA have been performed: the first is based on ANSAware; the second is based on BSD sockets. ANSAware provides a more sophisticated network programming paradigm than that provided by the BSD sockets model. In particular, it provides location-transparent binding based on resource matching and access-transparent communication between stub and server. Furthermore, the ANSAware nucleus provides a multi-threaded run-time environment over a wide variety of platforms including IBM PCs running MS-DOS. The main disadvantage of ANSAware is that the complexity of the software imposes a significant performance penalty. As CPU performance continues to improve, this performance penalty will become less significant.

The BSD sockets implementation provides a reasonable level of performance and is portable to a wide variety of platforms. The multi-threaded functionality of the BSD sockets `eserver` requires support from the SunOS LWP library and is therefore Sun-specific; a more portable implementation would be possible using a standard threads package such as POSIX `pthreads`. The main disadvantage of the BSD sockets implementation is that it lacks many of the services provided by the new generation of distributed environments such as ANSA.

Two changes to the transputer i/o libraries have been suggested to improve throughput for situations in which the transputer and server are separated by a network. The first is to increase the size of the client-side buffers to match the maximum packet

size supported by the link protocol; this minimises the impact of per-packet overheads. The second is to allow multiple request packets to be sent before waiting for the replies; this minimises round-trip delays thereby increasing server utilisation.

The BSD sockets implementation of the Enhanced Transputer Server Architecture is in the Public Domain. The software and documentation is available by anonymous FTP from `unix.hensa.ac.uk` in the directory `/parallel/software/iservers`. Contact the author (preferably by electronic mail) in case of difficulty.

References

[1] APM Ltd, The ANSA Reference Manual. APM Ltd, Poseidon House, Castle Street, Cambridge, CB3 0RD, UK, 1990.

[2] D. Comer, Interworking with TCP/IP. ISBN: 0-13-470188-7, Prentice-Hall International, 1988.

[3] L. Rizzo, PCserver: Networking PC-hosted Transputers. In: Transputer 92. ISBN: 90-5199-081-2, IOS Press, 1992.

[4] INMOS, IMS B300. INMOS, 1000 Aztec West, Almondesbury, Bristol, BS12 4SQ, UK, 1992.

[5] M. Morgan, WIServer. Available by anonymous FTP from unix.hensa.ac.uk in the directory /parallel/software/iservers, 1993.

[6] C.A.R. Hoare, Communicating Sequentail Processes. ISBN: 0-13-153289-8. Prentice-Hall International, 1985, pp. 133.