Berkeley Sockets on a SMPD Environment

Fredy João Valente
fvalente@icmsc.sc.usp.br

ICMSC-USP
Caixa Postal 668, São Carlos-SP-Brasil, 13560-970
FAX +55 162 749150

January 12, 1996

Abstract

No passado, sistemas de computação paralelos e distribuidos foram desenvolvidos seguindo diferentes caminhos. Recentes avanços nas tecnologias de redes de computadores e microprocessadores fez com que ambas areas se aproximassem de tal forma que atualmente elas podem ambas serem designadas e programadas de maneira similar como um sistema multiprocessador, usando-se um ambiente de programação comum (embora com diferentes níveis de granularidade e desempenho). Este artigo descreve o desenvolvimento de serviços de redes de computadores (UDP/IP) para um ambiente multiprocessador SPMD para que se possa promover uma transparente integração deste ambiente num sistema heterogeneo.

Abstract

Parallel and distributed systems have in the past evolved following different paths of development. Recent advances in networking and microprocessor technologies have brought them closer together and currently both areas can be regarded and programmed in a similar fashion as a multiprocessor system with a common programming environment (albeit with different levels of granularity and performance). This paper describes the development of networking services (UDP/IP) for a SPMD multiprocessor environment which is intended to promote a transparent integration of such a system onto a heterogeneous environment.

1 Introduction

In the continuous quest for an ideal general-purpose high-performance parallel computing environment many new computer technologies have emerged. The evolving of parallel computers into a scalable multicomputer architecture appears to be a solution to the von Neumman 'bottleneck' problem, making 'Teraflop' computers feasible. The architecture of scalable multicomputers is based on the interconnection of processor nodes through a high-bandwidth communication network. This has instigated the development of new programming paradigms such as message passing [18] and data-parallel languages [15].

Together with these developments, advances in computer network technology and communication protocols have facilitated distributed computing systems which greatly enhance the sharing of resources (such as printers, fileservers or the front-end computer of a parallel machine) and the integration of heterogeneous systems. Distributed systems are normally based on a client-server relationship and multitasking operating systems such as UNIX[27][14]. The combination of the features of distributed systems and the message-passing programming paradigm provided a framework for heterogeneous concurrent computation in networked environments. This in turn led to the development of products such as PVM (Parallel Virtual Machine) [31] and MPI (Message Passing Interface)[13].

Other developments in distributed systems area such as the advent of microkernel-based operating systems, for instance Mach [3] and Chorus [5], further contributed with new techniques such as the threads model [32] now used in concurrent computation on distributed systems. These developments have in a way encouraged some HPC (High Performance Computing) manufacturers into providing full UNIX on each node of a parallel machine. Among the recent examples of systems with this feature are the Meiko CS-2 [2] and the IBM SP-2.

Given this scenario towards providing UNIX for multiprocessors, in this paper I describe the provision of a UNIX-like networking services for SPMD multiprocessor machines which provides the portability advantages of UNIX without incurring in the inherent performance penalty. The resulting system is a low-cost multiprocessor environment incorporating standard distributed computing systems communication services. The environment is intended to provide a high level of transparency and flexibility and thus effectivelly promote the merging of parallel and distributed systems.

The demonstration system was developed based on transputers incorporating a network I/O node. The network software is modular enough to allow easy portability onto other hardware platforms. The system has been integrated onto a stand-alone SPMD environment in a way that all the processes can transparently access UDP-based network services on a LAN using the standard Berkeley sockets interface.

2 The Transputer Socket System

The Transputer Socket System (TSS) developed comprises a BSD-UNIX style socket library which can be used transparently by a parallel application to communicate with the Internet world using the UDP protocol. This provides the parallel application with potential access to all UDP-based local area network services (e.g. NFS (Network File System)). Transparency is accomplished by implementing the socket library as 'proxies'[1] accessing a message-passing system to communicate with a network subsystem server.

Motivation and Goals

Advanced techniques for programming scalable parallel architectures have the important objective of making parallel processing less explicit. Parallel extensions are being placed into existing software by moving the burden of parallel programming from the programmer to the compiler, for instance hiding from the programmer the data array distribution among processor elements of a parallel machine [15]. So far, the most often adopted methods for parallel programming have been the data-parallel [15] and SPMD approaches. Another major area of concern is related to the I/O requirements. According to DeBenedictis, a balanced I/O requires the I/O rate in Mbytes/sec, to equal the computing rate in Mflops/sec [12], which means that a scalable parallel machine requires a scalable I/O system for optimum balance.

System requirements – in order to bring the worlds of parallel computing systems and distributed computing systems together it has been decided that the system proposed so far must: (a) support access to a distributed IPC and to a parallel IPC mechanisms via a common API thus providing an environment that provides a standard framework for designing, implementing and integrating distributed and parallel software, (b) provide an API which is network-transparent on networks constructed of heterogeneous system components, (c) simplify and reduce coding effort when porting a native UNIX application onto the system by supporting a UNIX-like networking API on each process of a SPMD programming environment, (d) have a degree of independence from the SPMD programming environment in order to allow straightforward portability onto other similar systems, (e) be potentially scalable and portable across different hardware platforms.

The choice of the BSD-UNIX socket (discussed further in Section 4) as our networking API fulfills directly the first three requirements. Among the features provided by this API the following are specially important in this context:

- the BSD-UNIX socket API offers full compatibility with the UNIX and TCP/IP communication protocol domains
- it is relatively easy to support other communication domains
- the interface is independent of the operating system and hardware platforms
- even if the socket interface is not standardised by any institution, it is a *de facto* industry standard (e.g. Microsoft's WinSock interface is a variant of the BSD-UNIX socket interface [29]).

[1] A proxy is a local data object in the client whose interface is identical to the service interface, but implemented with marshaling stubs [4].

- reference implementations of parallel programming environments for heterogeneous systems such as PVM and MPI are based on the socket interface.

Alternatives such as the Transport Level Interface (TLI) or the Extended Transport Interface (XTI) could also meet the requirements met by the socket interface, however TLI/XTI can only be fully explored if STREAMS (a mechanism introduced with the UNIX System V Release 4 which essentially provides networking protocols and drivers embedded in the operating system core using software interrupts [29]) is provided, moreover TLI/XTI interfaces are more expensive to use than the sockets interface because they have a higher level of abstraction.

The latter two requirements are met by the design and implementation strategies adopted for this system which are discussed in the following sections.

3 System Architecture Design

Our underlying hardware platform is composed of a generic network of transputer nodes, one of which is attached to a front-end computer and another one to a local area network via an Ethernet connection as depicted in Figure 1.

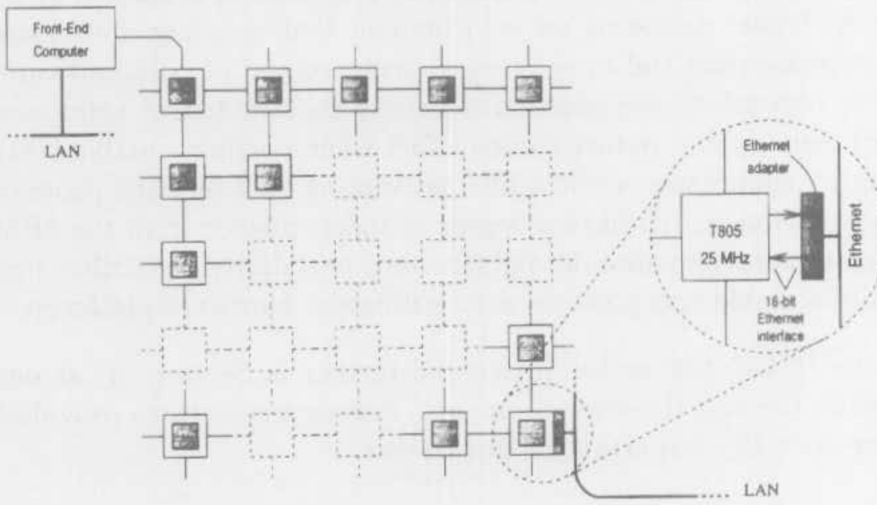

Figure 1: A general transputer-based parallel computing platform

Our protocol architecture provides connectivity between every processor present in the transputer machine and the local area network using UDP which can be accessed by the application on each node through a BSD UNIX-like socket interface. Transparency is achieved by building the interface on top of a message-passing environment giving the application running on each node the notion of a direct connection to the Internet, regardless of the node location or topology of the transputer network. The UDP/IP layered model of this system is shown on Figure 2. The main difference between the TSS (transputer socket system) and the traditional UDP/IP architectures relate primarily to the philosophy of the interconnection of the UDP layer with the Ethernet node and the UDP layer being distributed on all the nodes of the parallel application.

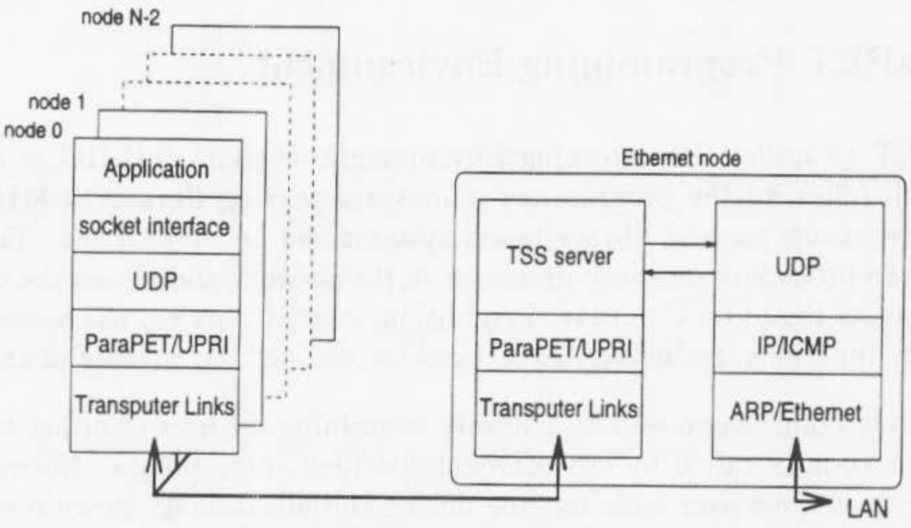

Figure 2: The TSS Protocol Architecture Model

Three major software components implement the above protocol architecture as depicted on Figure 3:

- The TSL (transputer socket library) which is available to the application domain provides an interface to the UDP protocol.
- The ParaPET/UPRI system which provides communication between processes on different nodes.
- The TSSNS (transputer socket system network subsystem) which is responsible for providing network services such as data send and receive, connection establishment, network address resolution (ARP protocol), network routing and packetisation/reassembly (IP protocol). The TSSNS is also responsible for controlling network sessions on different nodes by managing ports and sockets globally.

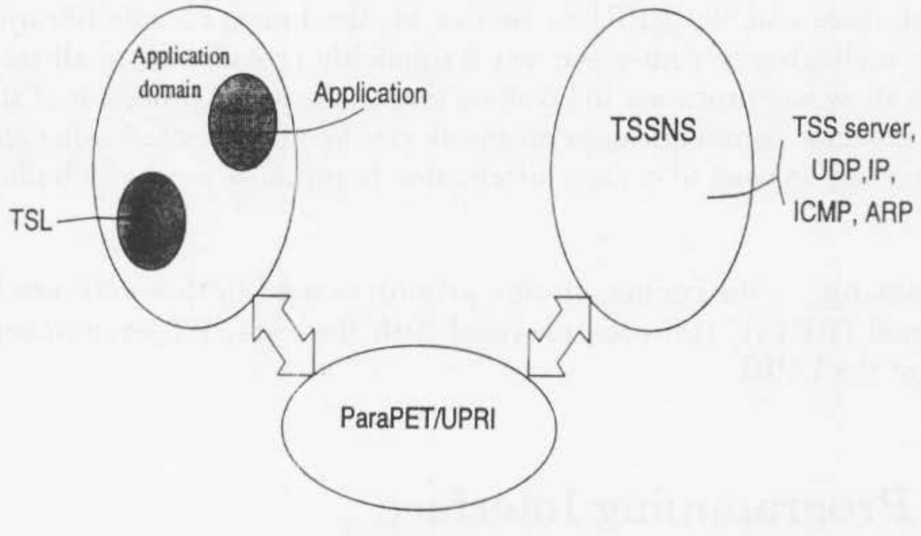

Figure 3: Software components of the TSS

The TSL and the TSSNS cooperate to manage network sessions using a lightweight request-reply (LWRR) protocol.

The ParaPET Programming Environment

The ParaPET (Parallel Programming Environment Toolkit) [11] [16] is a development toolkit which offers to the programmer a message-passing library SPMD model, upon which other message passing library-based systems can be constructed. The system supports a C `main` procedure on every processor in the network and allows the direct mapping of user interfaces based on C onto it. In addition, every C process has access to the standard C library functions, including instructions for device I/O, such as `printf` and `scanf`.

The `main` entry point is exposed by a library containing the user interface to the software, then the user code is called by environment-specified entry points. Normally, the ParaPET system loads one user code module during initialisation on every node available on the transputer machine, however if this is not sufficient for the application, a library for dynamic code loading is provided.

UPRI/VPI

The UPRI (Universal Packet Router Interface) [9] is a low-level interface between the communication network and a defined communication interface for distributed software. The system consists of kernel processes that access the communication resources of the underlying hardware (e.g. transputer links) and also library routines which are called by processes implementing communication protocols [9].

The VPI (Virtual Processor Interface) provides a message-passing model by directly exploiting the virtual channel facilities of the VCR (Virtual Channel Router) system [10]. The VPI forms the core of the ParaPET system in which processors in the transputer network are identified by VPI numbers (integers ranging from 0 upwards). The root processor (node attached to the front-end machine) is always numbered as 0. VPI numbers can be accessed by ParaPET primitives provided. The VPI exposes a low-level process scheduling interface and the UPRI as well as all the Inmos-specific libraries [11]. The VPI provides a client-server interface which implicitly provides all-to-all interconnection of channels to allow any processor to invoke action on any other processor of the transputer network, for instance communication protocols can be implemented using this feature or the mechanism can be used to convey information to initialise protocols built upon UPRI.

Message Passing – the communication primitives used in this work are based on the Reactive Kernel (RK) [1] [18] code provided with the ParaPET environment which are built on top of the UPRI.

4 The Programming Interface

The Berkeley sockets mechanism is first introduced and then the implementation of this interface for the TSS is described.

Berkeley Sockets

Sockets are endpoints of communication channels which were introduced for the first time in 1982 with the UNIX 4.1aBSD (Berkeley System Distribution) [14] as an IPC method between local UNIX processes. In the UNIX 4.2BSD release, sockets were used as a communication interface for UNIX processes to communicate with other processes (local or remote) in two different communication domains: UNIX and TCP/IP. Many network applications such as the first versions of the Sun NFS [30], the MIT X Windows system and the OSF DCE RPC [8] were developed using the Berkeley sockets interface [29].

The TSS Interface

The transputer socket system interface (TSSI) emulates the BSD UNIX socket interface by exporting a procedure call interface that is identical to the one exported by the latter. The interface resides in the transputer socket library (TSL) which can be linked to the application. Figure 4 lists the calls implemented in the TSL module.

```
int socket(int family, int type, int protocol);
int sendto(int s, char *msg, int msglen,
           int flags, struct sockaddr *to, int tolen);
int recvfrom(int s, char *buf, int buflen,
             int flags, struct sockaddr *from, int *fromlen);
int connect(int s, struct sockaddr *to, int tolen);
int send(int s, char *msg, int msglen, int flags);
int recv(int s, char *buf, int buflen, int flags);
```

Figure 4: The transputer socket system interface (TSSI) exports the standard BSD socket interface. These primitives form part of the transputer socket library (TSL).

Creating network sessions – The `socket` call creates a socket descriptor to represent a network session. The address family field specifies the communication domain in which communications are going to take place, for instance `AF_INET` should be used for the Internet domain, whilst `AF_TRANSPUTER` is used when performing communication between transputers using the TSSI. This is illustrated in the Figure 5. The TSS supports a connectionless protocol (UDP) which must be specified using `SOCK_DGRAM` when opening a socket. Finally, the protocol field selects the networking protocol to be used, which in our case is the `PF_INET` (Protocol Family InterNET). If the desired protocol happens to be the default protocol, it can be selected by using 0.

The `socket` library call performs a `request_socket` call to the TSSSERVER which will allocate a data structure representing an unconnected UDP session and then return an associated socket descriptor to the library call, which in turn returns it to the calling application. The uniqueness of the local endpoint is guaranteed and managed by the TSSSERVER. At this point a network session is established.

```
s = socket(AF_INET, SOCK_DGRAM, 0);
s = socket(AF_TRANSPUTER, SOCK_DGRAM, 0);
```

Figure 5: Creating Internet and transputer sockets.

Establishing connections – The act of invoking `connect` connects the local endpoint of a socket to a remote endpoint. For a connectionless protocol such as UDP, this action happens only locally. The application must supply an opened socket and a remote address which is specified using a Internet socket address or a transputer socket address (Figures 6 and 7 respectively). The `connect` library call will pack all information into a message and then contact the TSS server using a `request_connect` call.

```
struct in_addr {u_long s_addr;};          /* Internet address */

struct sockaddr_in {     /* socket address, Internet style */
    short      sin_family;
    u_short    sin_port;
    struct     in_addr sin_addr;
    char       sin_zero[8];
};
```

Figure 6: Internet socket address structure.

The TSSSERVER will literally copy the foreign Internet socket endpoint to the correspondent socket structure and mark the socket as connected. The advantages of using a connected socket is that there is no need to append the endpoint address to the message when doing a send operation thus saving time. All the address manipulation and packet header assembly will be handled by a lower layer software of the TSSNS.

```
struct sockaddr_tr {   /* socket address, transputer style */
    short sin_family;
    unsigned short t_node;          /* transputer node number */
    char sin_zero[12];
};
```

Figure 7: Transputer socket address structure.

Sending and receiving data – Although in the BSD socket interface there are ten different ways to handle data moving through a network session, the TSSI supports only four of them (`sendto, recvfrom, send, recv`) because they suffice for most applications. The scatter/gather versions of send and receive (`sendmsg, recvmsg`) which are normally used in UNIX systems to manipulate scattered data buffers such as MBUFS are not applicable to TSS due to the distributed memory nature of the system and also the TSSNS node does not provide support for DMA which is normally used in scatter/gather implementations.

Outgoing UDP datagrams are packed in a message and then sent to the TSS server by doing either a `request_sendto` or `request_send` call. Because the communication with the TSSSERVER is based on the synchronous blocking primitives of the ParaPET/UPRI, `sendto` and `send` are also blocking. However as soon as the TSSSERVER confirms receipt of the `request_send` call, the application `send` unblocks independently if the packet has already been sent down the network or if it is still in one of the TSSNS output queues waiting to be served. Summarising, the `send` operation on the TSS is only synchronous blocking from the application up to the TSSSERVER.

The library calls `recvfrom` and `recv` control the access to inbound UDP datagrams by first contacting the TSS server with a `request_receive` call which will cause the corresponding socket to be put into a waiting state. The `recvfrom` primitive then blocks itself on a ParaPET/UPRI receive primitive until the data arrives from the TSSNS node when the data contained on an UPRI buffer is copied onto the application buffer.

All the operations on transputer sockets are entirely handled on a one-to-one basis without any interference from the TSS server. Send operations block until the entire message has left the sending processor. Receive calls will be in a blocked state until a complete datagram has been read in. If an application wishes to perform communication between transputer nodes using Internet sockets, it can be realised using the Internet loopback address (127.0.0.1) which will cause the datagrams to be routed through the TSSNS.

Address manipulation – Internet addresses can be easily manipulated using the `inet_addr` instruction provided by the TSL, which is also a system library standard provided by UNIX systems (see example in Figure 8). Other Internet address manipulation instructions such as `gethostbyname` could be implemented by contacting the local DNS (Data Name Server) if necessary.

For transputer sockets, the `t_node` field of `struct sockaddr_tr` (Fig. 7) must contain the destination node number on send. This field will contain the source node number on receive.

```
#include    <tsock2.h>          /* TSS header file          */
#include    <t_in2.h>           /* TSS Internet header file */
...
char *caesar = "152.78.66.170"; /* caesar Internet number */
struct sockaddr_in sin;
...
sin.sin_addr.s_addr = inet_addr(caesar);
sin.sin_family = AF_INET;
sin.sin_port = IPPORT_ECHO;  /* to access the echo daemon */
```

Figure 8: Assigning a Internet address to a socket.

5 The TSS Network Subsystem

The transputer socket system network subsystem (TSSNS) comprises the TSS server and the networking software (UDP, IP, ICMP, ARP) as previously depicted in Figure 3, running entirely on a dedicated transputer node that has an interface to Ethernet.

TSSNS Software Organisation

The TSSNS consists of processes that have access to the networking resources provided by an Ethernet interface and to the communication facilities of ParaPET/UPRI. The idea behind the design of the TSSNS was to provide a modular network subsystem that could be integrated into different parallel programming environments. For this, in the TSSNS the network layer and the data link layer were implemented as kernel processes which are completely independent of the parallel programming environment.

The interaction between the TSL calls and the networking services is controlled and performed by the TSS transport and control (TTC) layer, as illustrated in Figure 9.

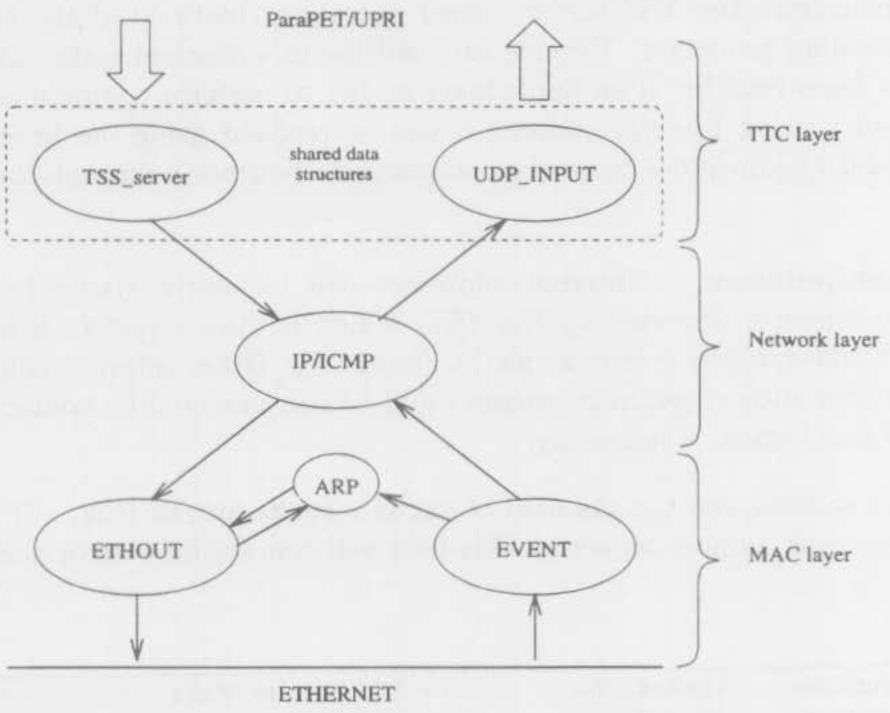

Figure 9: The TSSNS Communication Software Structure

The TTC layer communicates with the application threads on one side by means of UPRI communication primitives and with the network layer on the other side using network packet queues. Refer to Section 6 for a complete description of the TCC layer. The following subsections outline the implementation of the network interface and the networking software (up to IP).

The Network and Data Link Layers

The network (IP/ICMP) and data link (Ethernet/ARP) layers of the TSS are based on the TCP/IP protocol suite implementation described by Comer & Stevens [6]. The differences are either necessary modifications to adapt the code to different hardware or improvements achieved by porting the code to the threads environment of transputers. These are outlined in the following sections.

The whole of the TSS system is written using the C language for various reasons. The ParaPET/UPRI was developed in C as well as the network protocol software over which the implementation was based. The compiler used in this work, the Inmos C D4314a, has good support for low-level communications (providing semaphores, process management and IPC via CSP channels).

The Transputer-Ethernet Interface

The transputer-Ethernet interface developed in this work, shown in Figure 10, comprises a Inmos T805 transputer with 4 Mbytes of DRAM and an interface to Ethernet. The transputer type and amount of memory have been chosen in accordance with the rest of our hardware platform which is composed of T800 transputer nodes with the same amount of memory (4 Mbytes). For the Ethernet interface, a pragmatic approach by developing an interface to a PC-AT bus has been taken so that inexpensive 'off-the-shelf' Ethernet cards, such as the Western Digital 8013a, can be used. In addition the interface can directly accept other EISA devices such as graphics cards.

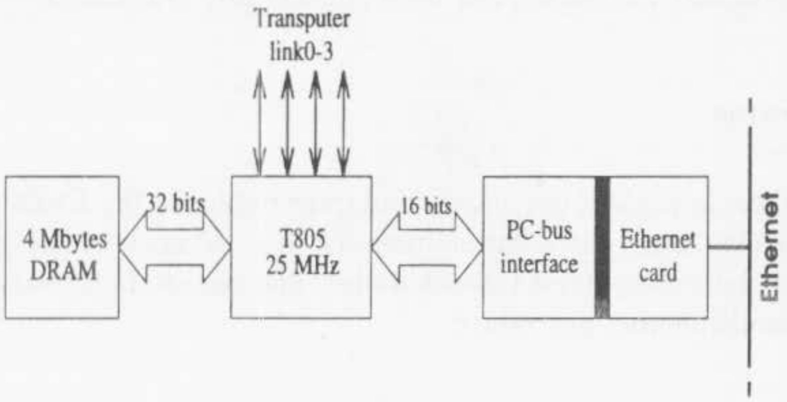

Figure 10: The Transputer-Ethernet Interface

The interface to the PC-bus is mapped into memory, thus leaving all the transputer links free to be interconnected with other nodes. The advantage of this approach is that this 'special' node can be placed anywhere in the transputer network just like an ordinary node. A complete description of the interface can be found in [34].

Event Driven I/O – Usually, the network interface uses a interrupt mechanism that will cause a jump from normal processing to a *packet driver* [6]. Interrupts have been implemented by using the transputer EVENT line, which is an input pin that is connected to a channel word in memory. Our event thread waits on a `ChanIn` instruction until the Ethernet controller generates a interrupt request which will cause the transputer EventRequest line to be asserted (see hardware details in in [17]).

Sending Datagrams

When the TSS server wishes to send an UDP datagram it invokes the procedure `udpsend(fip, source_port, dest_port, pep, dlen, docksum)` which will send the datagram of length `dlen` in network buffer `pep` to a destination IP address `fip` after filling the UDP header (`[source_port, dest_port, dlen, checksum]`) in the datagram. UDP checksum can be switched ON/OFF by toggling `docksum`. The datagram is then sent by calling the IP protocol entry point `ipsend` which places the network buffer in an output IP queue, increments an IP counting semaphore and returns to the TSS server which will then be free to attend another request. From this point downwards, the network and data link layers take over responsibility for the data.

The IP process fills in the IP header (see format in [6] and [29]), looks up a route for the datagram in the IP routing table, fragments the datagram if it is too large for the output interface (Ethernet), checksums the IP header and hands the datagram to the ETHOUT process by passing a pointer to the network buffer using the `DirectChanOutInt` (outputs an integer on a soft CSP channel) command.

The ETHOUT process, which works as a packet driver, blocks on `DirectChanInInt` until a network buffer address is read (when a network packet is ready to be written). ETHOUT fills in the Ethernet header, copies the packet to the Ethernet adapter and blocks until the EVENT process signals that the packet has been successfully transmitted to the Ethernet.

Receiving Datagrams

Upon receiving a Ethernet packet, the network adapter unblocks the EVENT process which in turn copies the packet from the adapter memory to a network buffer which resides in the main memory of the transputer-Ethernet node. The packet is forwarded upwards by calling the input demultiplexing procedure.

Input Demultiplexing – is done by switching the Ethernet packet type field of the Ethernet header, as illustrated in Figure 11. ARP requests (`EPT_ARP`) are serviced immediately by `arp_input` routine. Packets containing IP datagrams (`EPT_IP`) which were not directly sent to our IP address and are not IP broadcast addresses are discarded. This check which is not normally done in traditional network software implementations, was included here because it was noticed that the IP process was spending too much time handling (and discarding) spurious network packets generated by software such as Microsoft Windows.

The processing of ICMP input packets was optimized by calling `icmp_input` directly from this point skipping the IP process on input. This has the advantage of avoiding queuing the packet twice (on the IP input and output queues) because the resulting packet is placed

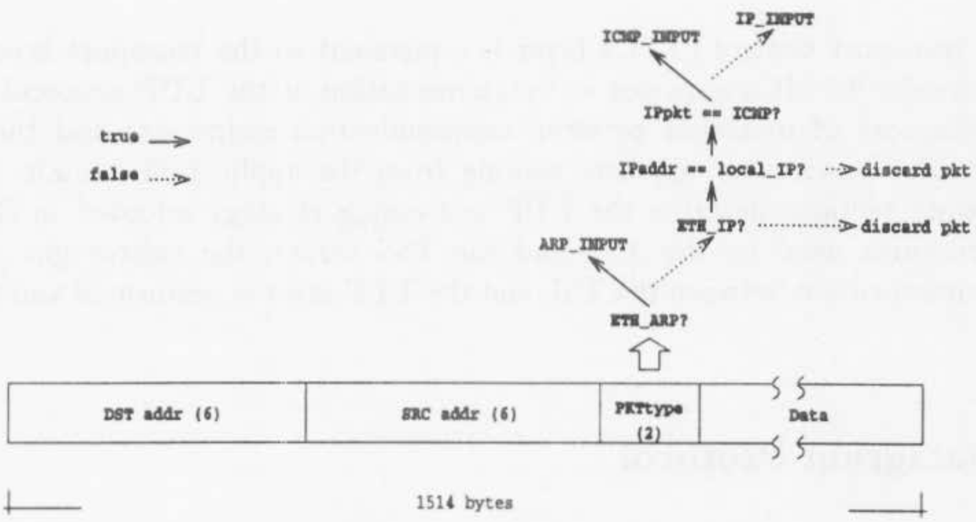

Figure 11: Demultiplexing Ethernet input packets

directly on the IP output queue after ICMP processing, thus lessening the load on the IP process. The optimisation improves the ICMP echo request response time by more than 20 %.

Ordinary IP packets are placed in the IP input queue. After checking the IP header and datagram length for validity, the IP process puts packets containing UDP datagrams into the UDP input queue. Valid UDP datagrams are sent to the appropriate socket by the UDP_INPUT process.

Internet Protocol on Transputers

The Internet Protocol implementation adopted in this research executes as a single independent process. This strategy has been adopted because it simplifies the porting of the code and also because a multithreaded approach would require substantial modifications in the protocol structure. Packets coming from lower and upper layers wait to be served by IP in the IP input queue(s) and IP output queue(s). The former contains packets coming in from the network while the latter contains locally-generated traffic. Although the IP software has been tested with only one input and one output queues, there is no restriction to the number of IP queues. When all the queues are empty, the IP process blocks on a counting semaphore until a packet requesting IP processing is placed in one of the queues. Fairness is achieved by serving packets in a round-robin fashion. After IP processing, packets are either placed in the UDP input queue or are sent to the ETHOUT process via a CSP channel.

The IP implementation allows direct broadcasting which means delivery to all hosts and gateways on the specified network. This feature increases the usability of the interface. A full discussion on this topic can be found in Comer & Stevens [6].

6 TSS Transport Control Layer

The TSS transport control (TTC) layer is equivalent to the transport layer of the OSI reference model [6]. It comprises an implementation of the UDP protocol (responsible for the transport of messages between communication endpoints) and the TSS server which provides services to requests coming from the application domain via the TSL. The following sections describe the UDP processing strategy adopted in this work, the socket structures used by the TSL and the TSS server, the lightweight protocol over which communication between the TSL and the TTC layer is performed and the data path utilised.

User Datagram Protocol

The User Datagram Protocol (UDP), specified in RFC 768 [24], is a connectionless transport protocol addressed via port numbers. UDP adds only port addressing and data checksum to the IP layer as depicted in Figure 12. Unlike reliable transport protocols such as TCP [26], UDP does not provide any form of acknowledgement or other reliability mechanism. However, this simplicity makes UDP very efficient and suitable for high-speed applications on the LAN domain, where data integrity can be trusted to the underlying network technology such as Ethernet.

For these reasons, the UDP protocol was adopted for the purpose of providing networked services (such as NFS) to a parallel application.

```
struct udp_h{
    unsigned short src_port; /* source UDP port number      */
    unsigned short dst_port; /* destination UDP port number */
    unsigned short d_len;    /* length of UDP datagram      */
    unsigned short chksum;   /* UDP cheksum (0 => none)     */
};
```

Figure 12: The UDP header structure

Ports and Input Demultiplexing - protocol port numbers are used by applications to identify the endpoints of communication. A client-server applications wishing to communicate must specify the destination port number as well as the local port number in addition to the foreign IP address.

In order to allow applications running on different nodes of the transputer machine to communicate with multiple remote sites simultaneously, while ensuring that the input demultiplexing is simple and efficient, the input selection is done using the destination port number only (Figure 13 illustrates a case when the UDP port 2051 has been mapped to an application socket on node 0, port 2072 to node 1 and port 2090 to a node N). One of the advantages of this style is that all the input datagrams can be placed on the same input queue. The main drawback of this scheme is that erroneously addressed datagrams cannot be filtered by the system. The alternative way (normally used in UNIX implementations) of demultiplexing using the source and destination port numbers as well as the IP source number is sometimes inefficient when a single application needs to communicate with more than one remote application simultaneously. In such circumstances the system must allocate one input queue for each of them and also provide a mechanism such as the UNIX `select`, to control the I/O activity on the queues. The TSS UDP selection mechanism can be based solely on the UDP port number because of the filtering mechanism employed on the Ethernet input demultiplexing mechanism (refer to Figure 11) which only allows IP packets directed to the local IP address and IP broadcast addresses to enter the system.

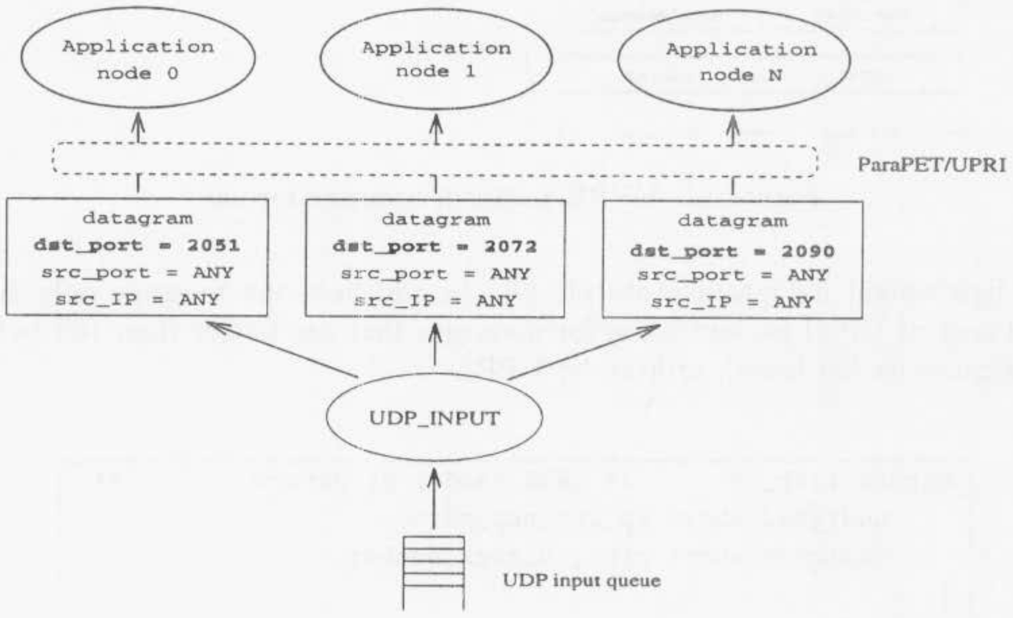

Figure 13: Demultiplexing input UDP datagrams

LWRR Protocol

The LightWeight Request Reply (LWRR) Protocol has been specifically devised in order to provide efficient communication between the TSL and the TCC layer. The LWRR is a header-based datagram protocol which makes use of the request/reply principle used in client-server programming. The protocol message format illustrated in Figure 14 has three main fields: the LWRR header which is always present, the Internet socket field and a UDP datagram field which can be present or not depending on the request. A message using the LWRR protocol is referred to as a 'TSS datagram'.

The message format has been designed using two main considerations: (a) the LWRR header length (4 bytes) plus the `sockaddr_in` field (16 bytes) equals the length of the IP header. As the TSS datagram is defined as a structure, shown in figure 15, this has the advantage of mapping directly into the IP and UDP header templates in the Ethernet packet, so the TSS datagram can be directly copied into a network buffer from the first byte of the IP header with most of the parameters already in place saving a considerable amount of time positioning protocol fields, (b) the TSS datagram length up to the end of the UDP header totals 28 bytes, fitting into the minimum UPRI packet which is 32 bytes. This minimum length imposed by the UPRI has been reserved for use in the

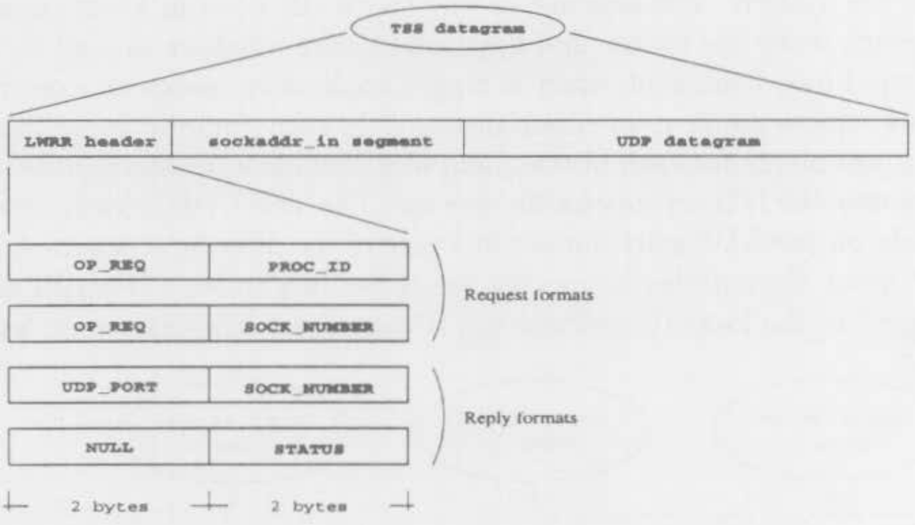

Figure 14: LWRR protocol message format

design of lightweight datagram protocols [9]. In addition, the message only suffers the additional cost of UPRI packetisation for messages that are longer than 160 bytes which is the maximum packet length utilised by UPRI.

```
struct lwrr_h{       /* LWRR header structure              */
    unsigned short op_req_udp_port;
    unsigned short proc_id_sock_number;
};
/* Request format */
#define OP_REQ      op_req_udp_port
#define PROC_ID     proc_id_sock_number
/* Reply format */
#define UDP_PORT    op_req_udp_port
#define SOCK_NUMBER proc_id_sock_number
#define STATUS      proc_id_sock_number

struct tss_dgram_h{ /* TSS datagram header structure */
    struct lwrr_h tss_h;
    struct sockaddr_in sock_inet;
    struct udp_h pkt_udp_h;
};
```

Figure 15: The TSS datagram header structures

When sending a request from a generic transputer node to the TSS server, the OP_REQ field of the LWRR header request format, depicted in Figure 14 must contain one of the LWRR requests codes (refer to [34] for details). The second field must contain the processor identification number (PROC_ID) which can be recovered by using the ProcId() primitive provided by the ParaPET to access VPI numbering (Section 3), or the socket number (SOCK_NUMBER) over which the operation is to be performed. PROC_ID provides essential information, since the TSS server associates it to a socket enabling replies to be sent direct to the proper destination client processor nodes. Once a client has already had a socket allocated to it by the TSSNS node, most of the subsequent requests are done using the socket number.

7 Summary and Conclusions

The system described in this paper is operational and being used in another PhD research at Southampton University, United Kingdom. The TSS supports the access to networking services based on the UDP protocol on each processor node of the transputer machine. This is achieved through a BSD-UNIX-like sockets library referred to here as TSL (Transputer Socket Library) which among other advantages provides a transparent access to UDP in the same fashion as in UNIX. A transparent access means that the application is completely oblivious about the mechanism employed to get the message to the destination socket. This feature eases the porting of UDP-based applications and UNIX network services such as TFTP (Trivial File Transfer Protocol) [29], RPC and NFS[2] onto the system.

The mechanism for UDP/IP sockets support on the TSS is based on a client-server interaction between the TSL on each processor node and a network server on an I/O node.This involves the marshaling of the socket operation parameters into a TSS datagram which is sent to the TSSNS (TSS network subsystem, which includes the network server, residing on the I/O node) where all the UDP/IP processing and socket management are handled. Control over the TSS datagrams is achieved via the LWRR protocol which was developed for this project.

Although performance has not been addressed in this paper, the system described here has been extensively tested and validated by the implementation of a SPMD NFS which is described in [34]. Bandwidths in the 100 Kbytes/sec order have been achieved during simultaneous NFS file transfers for 3 different processor nodes.

The present version of the TSS provides enough support for integrating a transputer-based multicomputer into a heterogeneous computing environment, considering that communication in these environments is usually supported by message-passing systems such as PVM or MPI (which are normally based on UDP in the LAN domain). The term metasystem has recently been introduced to describe such environment [35].

References

[1] Athas W.C., Seitz C.L., *Multicomputers: Message-Passing Concurrent Computers*, IEEE Computer, Vol.2, No.4, pp. 9-24, Jul/Aug-1988.

[2] Barton E., Cownie J., McLaren M., *Message Passing on the Meiko CS-2*, Parallel Computing, No.20, pp.497-507, 1994.

[2]NFS is potentially portable to any kind of system, however all the reference implementations are based on the Berkeley sockets interface.