Abstract:

This paper describes an implementation of a software interface designed for use from within UNIX* application programs for network communications. This interface provides the abstraction of possibly very large datagrams, supports 'scatter-gather' facilities, and maintains standard network addresses consisting of ⟨host number; port number⟩ pairs. This interface has been provided on different data communication facilities allowing uniform program access to those facilities. The particular implementation developed for a local area network is described and performance results obtained are examined and compared with those obtained from a conventional datagram interface to that network. The results obtained confirm that the abstraction of very large datagrams enables the construction of an efficient mechanism for process-to-process communications over the network that is not only more convenient, but also has significant performance advantages over the use of conventional (small) datagrams.

*UNIX is a Trademark of Bell Laboratories.

A Communication System Supporting Large Datagrams

on a Local Area Network

By

A. Linton and F. Panzieri

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick  
Number 193  
May, 1984

© 1984 University of Newcastle upon Tyne. 
Printed and published by the University of Newcastle upon Tyne, 
Computing Laboratory, Claremont Tower, Claremont Road, 
Newcastle upon Tyne, NE1 7RU, England.
A communication system supporting large datagrams on a local area network. [By] A. Linton and F. Panzieri.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1984.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 193.)

Added entries

PANZIERI, Fabio.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Dewey (18th): 001.64404
U.D.C. 519.687

Suggested classmarks (primary classmark underlined)

Suggested keywords

DATAGRAMS
LOCAL AREA NETWORKS
UNIX

Abstract

This paper describes an implementation of a software interface designed for use from within UNIX* application programs for network communications. This interface provides the abstraction of possibly very large datagrams, supports 'scatter-gather' facilities, and maintains standard network addresses consisting of <host number; port number> pairs. This interface has been provided on different data communication facilities allowing uniform program access to those facilities. The particular implementation developed for a local area network is described and performance results obtained are examined and compared with those obtained from a conventional datagram interface to that network. The results obtained confirm that the abstraction of very large datagrams enables the construction of an efficient mechanism for process-to-process communications over the network that is not only more convenient, but also has significant performance advantages over the use of conventional (small) datagrams.

*UNIX is a Trademark of Bell Laboratories

About the author

Mr. Andy Linton joined the Computing Laboratory in 1983 as a Research Systems Programmer.
Dr. Fabio Panzieri joined the Computing Laboratory at the University of Newcastle upon Tyne in June 1979, where he is currently a Research Associate.
A Communication System Supporting Large Datagrams on a Local Area Network

A. Linton and F. Panzieri

The University of Newcastle upon Tyne
Computing Laboratory
Newcastle upon Tyne NE1 7RU

ABSTRACT

This paper describes an implementation of a software interface designed for use from within UNIX* application programs for network communications. This interface provides the abstraction of possibly very large datagrams, supports "scatter-gather" facilities, and maintains standard network addresses consisting of <host number; port number> pairs. This interface has been provided on different data communication facilities allowing uniform program access to those facilities. The particular implementation developed for a local area network is described and performance results obtained are examined and compared with those obtained from a conventional datagram interface to that network. The results obtained confirm that the abstraction of very large datagrams enables the construction of an efficient mechanism for process-to-process communications over the network that is not only more convenient, but also has significant performance advantages over the use of conventional (small) datagrams.

Introduction

This paper describes an implementation of the Uniform Datagram Service (UDS) interface proposed in [Panzieri83]. This interface (i) is characterized by primitive operations for sending and receiving possibly very large datagrams, (ii) provides facilities for gathering data for output from, and scattering data in input to, non contiguous areas of the application's memory, and (iii) uses a simple network addressing scheme consisting of <host number; port number> pairs.

This interface is intended for use from within UNIX application programs for communication in an heterogeneous communications environment consisting of, for example, local area networks, packet switching

*UNIX is a Trademark of Bell Laboratories.
networks, and asynchronous lines. The software supporting this interface can be incorporated into the device drivers providing access to the networks to which a UNIX machine is directly connected; thus, application programs will be provided with a uniform program access interface to those networks. In general, the existence of multiple networks will be visible at the application level. (This is analogous to the way in which, in UNIX, application programs can access different disk units using the same set of UNIX system calls.) However, the interface can be used alternatively, or additionally, for access to one or more internetworks, i.e. one or more "virtual networks" which hide the existence of multiple physical networks.

The UDS interface has been provided, in our Laboratory, principally for use by the Newcastle Connection [Brownbridge82], a software subsystem that allows the construction of a distributed system out of a number of physically interconnected UNIX (or UNIX look-alike) systems. This distributed system, named UNIX United, is equivalent, both at the user and the program level, to a single-processor UNIX system.

In order to assess the viability of the UDS interface, implementations of this interface have been developed for different hardware communications technologies. These implementations include one for Cambridge Ring [Wilkes79] interrupt-driven interfaces, one for Cambridge Ring Direct Memory Access (DMA) interfaces, and one for RS232 interfaces to asynchronous lines; an implementation for Ethernet [Metcalfe76] interfaces providing ECMA transport service [ECMA81] is currently being investigated, and it is hoped, in future, to consider the use of an X25 wide area network.

This paper describes our implementation developed for the Cambridge Ring interrupt-driven interface and presents some performance results. These results, compared with those obtained from the original more conventional datagram interface to the Ring, show that the UDS interface noticeably improves the performance of communications between application processes, particularly when large data objects are exchanged. This improvement has been obtained without compromising the reliability of the communications.

This paper is organized as follows. The next section summarises the design of our interface; section 2 describes the structure and the implementation of the Ring driver that supports that interface; section 3 discusses the performance results obtained; finally, section 4 provides some concluding remarks.

1. Interface Summary and Overview of Design Issues

The design of the UDS interface, the semantics of its primitives, and their adequacy for a large class of distributed applications of UNIX have already been discussed in [Panzieri83]. However, for the sake of completeness, we provide a brief summary of this interface and review the major issues that motivated its design.
1.1. Interface Primitives

The following six primitives characterize the UDS interface. In keeping with the existing UNIX I/O calls, they are all synchronous operations, asynchronous behaviour requiring explicit use of multiple processes. As we shall see later, they have been implemented using standard UNIX `ioctl` calls rather than as new system calls. Networks can be conveniently represented within the UNIX file system by at least one special file [Ritchie74], conventionally held in the directory "/dev"; in the following, the parameter "netfd" is a normal UNIX file descriptor obtained by opening a network special file.

(i) **long receive(netfd, source, pbuff, timeout)** - receive a datagram. The source address of the datagram is made available in the parameter "source". The datagram is scattered to the set of buffers specified by "pbuff"; "timeout" indicates how long the process invoking `receive` is willing to wait for data.

(ii) **long send(netfd, destination, pbuff)** - send a datagram to the network address indicated by "destination". Data are gathered from the set of buffers specified by "pbuff".

(iii) **lto(r(netfd, raddr, laddr)** - convert logical network address "laddr" to real network address "raddr" for network "netfd";

(iv) **rtol(netfd, laddr, raddr)** - convert real network address "raddr" to logical network address "laddr" for network "netfd";

(v) **netget(netfd, pnet_if)** - general parameter enquiry function for network "netfd", via data structure "pnet_if";

(vi) **netset(netfd, pnet_if)** - general parameter-setting function for network "netfd", via data structure "pnet_if".

In essence, the UDS interface allows the exchange of variable length data units, of a maximum size of 2 G-bytes, from a set of non-contiguous areas of memory at a source network address to a set of non-contiguous areas of memory at a destination network address. Each of these data units is transmitted independently of all the other units; thus, each unit can be thought of as a "datagram" [Watson81]. Whether or not the underlying networking facilities actually provide a datagram service is a separate issue that can be dealt with by appropriately mapping the UDS primitives into those provided by the underlying network, as introduced below.

1.2. Design Issues

In order to maintain the semantics of all the UNIX system calls across the UDS interface, it is critical that the software implementing this interface be incorporated within the UNIX drivers providing network access. The structure of a driver can be thought of as constructed out of (i) a network-specific component, which implements network-specific functions (e.g. network-specific routing and flow control policies), and maintains a network-specific interface, and (ii) an Adaptor, a software
component which hides the network specific interface by mapping it into the UDS interface. (In practice, much of the actual network software might be implemented by a front-end processor; in such a case the actual UNIX driver will implement the UDS interface and some appropriate protocol for communications with the front-end.) The major responsibilities of an Adaptor are summarised below.

**Fragmentation and Reassembly:** One of the major objectives of UDS is the removal of these tasks from application processes since they may impose extra complexity and require a large number of system calls.

**Scatter-Gather Facilities:** These remove the need for copy operations at the application level when, for example, application-dependent protocols are implemented.

**Address Mapping:** Different network-specific interfaces are likely to use different address formats; the UDS interface instead provides a single, uniform address format, which can be used on each network by the application software. Each Adaptor maintains on the UDS interface a set of "logical" addresses that it maps into the set of "real" ones on the network-specific interface beneath it. The addresses provided by the UDS interface consist of pairs <host number; port number>. Each of these pairs identifies the address of a host computer on a network and a sub-address within that host. The mapping between logical and real addresses performed by an Adaptor can be obtained by the application, if required, by invoking the primitives `ltor` and `rtol`. The former converts a given logical address into its corresponding physical address, the latter converts a physical address into its corresponding logical address.

**Well-known Addresses and Signatures:** A process can obtain a port number thus enabling it to receive data addressed to that port. A port number can be either generated dynamically at the user’s request, or it can be explicitly selected, within a fixed range, by the user process, and communicated to the driver. The set of port numbers included in this fixed range ([0..999] in our implementation) constitutes the set of "well-known" addresses which can be statically allocated to specific processes. These addresses, can be used within a distributed application to make available specific servers, such as file or name servers. As distributed applications may require the use of "signed" messages, particularly in order to communicate with servers at well-known addresses, Adaptors are constructed so that they recognise well-known addresses and sign all the messages to such addresses with the "Signature" (see section 2) of the transmitting processes. Thus, application software is prevented from attempting any forgery of the signature.

**Exception Reporting:** Finally, Adaptors are responsible for indicating to the applications that a fault has occurred. The set of possible exceptions which may occur on different networks can be standardized, so that the task of the Adaptor is, where necessary, to map any network-specific exceptions into an appropriate standard exception. All UDS primitives may return with an error indication,
in the style of UNIX system calls. To allow the application software to discover the particular network-specific exception that has occurred, Adaptors make available a variable named "ud serr" which can be examined, like the UNIX variable "erno", by the application software, when an exceptional response occurs.

2. The Cambridge Ring Driver and the Adaptor

The Cambridge Ring is represented by a set of special files, as mentioned previously. A pair <major device number; minor device number> is associated with each such file; the "major device number" identifies the Ring driver; the "minor device number" is used by the driver to identify a particular Ring I/O port.

The original Ring driver available to us allows applications to operate on Ring special files using the same primitives as for ordinary disk files. The file descriptor obtained by opening a Ring special file is used by the application program to perform any further operations on the Ring. For example, ioctl operations provide the driver with the destination address of data to be transmitted, and write and read operations can be used to transmit and receive datagrams. However, the maximum size of a datagram that an application can transmit in a single write operation is of 512 bytes, because of limitations internal to the implementation of the driver (see below).

The Ring driver is implemented in two distinct parts. The "top" part, which we have termed Ring High, provide the applications with open, close, read, write, and ioctl operations. The "bottom" part, termed Ring Low, comprises the interrupt routines which implement the physical Ring I/O, and maintains a pool of system buffers (of 512 bytes each, hence the limitation above) used to store temporarily user data being transferred between the user memory and the Ring. Communication between these two parts is event-driven; to this end, UNIX provides internal routines (e.g. "sleep(event,...)", "wakeup(event)") to indicate the occurrence of some particular event, such as "start transmission" and "data received". Thus, the interface between these two parts can be thought of as characterized by (i) the pool of system buffers maintained by Ring Low, and (ii) a limited set of operations for acquiring and releasing those buffers, and notifying occurrences of events. In essence, in transmission for example, Ring High copies a user buffer of at most 512 bytes into one of the system buffers provided by Ring Low, and requires Ring Low to "start transmission"; in reception, Ring Low places a received block into a system buffer associated with a destination port, and notifies Ring High of the occurrence of this event. Ring High, in turn, copies the received data from that system buffer into a user provided buffer.

In addition, Ring Low implements the Cambridge Packet Protocol [Needham82]. This protocol allows the transmission and reception of variable length blocks of data (i.e. Packets), of at most 512 bytes in our implementation, between processes distributed on the Ring. The Packet Protocol defines also a simple naming scheme such that data can be transmitted to, and received at, specified "port numbers". Port numbers are implemented as integer-valued names defined in a name space.
local to each machine. A process can obtain a dynamically generated port number from the driver, or can select a specific port number in a fixed range and communicate it to the driver, thus enabling it to receive datagrams addressed to that port number.

2.1. The Adaptor

In order to support the UDS interface one strategy is to replace Ring High with appropriate software to map the UDS interface into the Ring Low interface; however, as we needed to support existing programs as well as ones using the UDS interface, we have found it convenient to structure our implementation as depicted in Figure 1. This implies that both the UDS interface and the Ring High interface are available to application programs and, in the implementation being described, intermingled use of these two interfaces by the same application is not prevented (see below).

```
UDS Primitives       Ring High Primitives
--------------------|-----------------------|
|                    | UNIX Kernel Interface
| UDS Adaptor| Ring High |  | Ring Low Interface
|            |          |  | Ring Hardware Interface
```

Figure 1: Ring Driver Incorporating UDS Adaptor.

The sending and receiving adaptors communicate using a simple protocol for transmission of UDS datagrams. In summary, UDS protocol data units [Zimmermann80], consisting of a variable length header and (possibly fragments of) a user datagram, are encapsulated in the data area of the packet provided by the Ring Low interface; the layout of a UDS unit is depicted in Figure 2. The UDS protocol imposes a minimum overhead of 2 bytes and, as the Ring Low Packet data area has a maximum size of 512 bytes, fragmentation and reassembly of application datagrams always occurs when the size of those datagrams is greater than 510 bytes. A UDS protocol data unit may contain further header information when an application datagram is larger than 510 bytes, and/or when its destination address is a "well-known" address. This additional information consists of a 4 byte "Length" field, to indicate the length of a datagram of up to 2 G-bytes, and an 8 byte "Signature" field, containing the real and effective user and group identifiers of the application process originating the datagram. The presence or absence of the "Length" and "Signature" fields in a UDS protocol data unit is indicated by the setting of the flags in the first word of that unit; if these fields are not present, the H1 part of the UDS unit contains application data.
Large application datagrams are transmitted as a sequence of Packets, each containing a single UDS unit. On the Ring, problems of out-of-order delivery of units do not occur; however, errors due to loss or corruption of UDS units are possible. Thus, the UDS protocol incorporates simple measures to detect the possible loss or corruption of a UDS unit in a sequence. No recovery measures are implemented within our Adaptor to cope with those errors; rather, when such an error is detected, the Adaptor raises an appropriate exception at the application level.

The following two practical problems were encountered during the implementation of our Adaptor.

(i) As the UDS interface allows processes to transmit and receive datagrams of up to 2 G-bytes in a single operation, it is undesirable that a process performing a large data transfer hold exclusive access to the network for the entire duration of that transfer. Thus, a choice must be made as to how to multiplex network access amongst distinct user processes.

(ii) With our UNIX systems and current hardware configuration, application processes can transmit data at a much faster rate than that at which they can receive data; as a result, a very high data-loss rate may be experienced at the application level (see section 4), if appropriate measures are not taken within the Adaptor.

Problem (i) has been solved by multiplexing the network access on a per data unit basis. That is, a user process transferring a datagram
holds exclusive access to the network for the duration of the transfer of a UDS protocol data unit of up to 512 bytes. Then, a re-allocation of the processor may occur allowing a different process to obtain network access.

The solution to problem (ii) has been to implement a simple flow control mechanism. The correct reception of each UDS protocol data unit causes the receiving Adaptor to scatter the received data to the buffers provided by the application process, and send an acknowledgement to the originating Adaptor, requesting the transmission of the next unit. Needless to say, the UDS unit containing the last fragment of an application datagram causes the receiving Adaptor to send the acknowledgement and to terminate the application’s receive operation. The Adaptor executing a send operation terminates normally when the acknowledgement to the last UDS unit is received. Thus, normal termination of the send operation indicates that the transmitted datagram has been scattered to the buffers in the memory of the destination application process. The acknowledgement to each UDS unit transmitted is expected within a timeout period (of 50 clock ticks in our implementation); lack of acknowledgement causes abnormal termination of the send operation. (Note that the particular fragmentation and flow control strategies we have adopted may not be the most appropriate on other networks.)

It is worth mentioning that our particular implementation, in which the pre-existing Ring High primitives are still available, may be open to threats from malicious users. It is possible for such a user to forge a data object identical to a UDS protocol data unit, and to transmit that object using the Ring High primitives, thus possibly causing considerable confusion.

Coping with this sort of problem was beyond the scope of our exercise; however, in order to maintain both the UDS and the Ring High interface and to provide on the UDS interface a similar level of security to that provided on the Ring High interface, a simple solution suggests itself. The Packet Protocol specifies a fixed header bit pattern to identify the beginning of a valid Packet; Ring Low could be modified so as to use a Packet header bit pattern other than that specified by the Packet Protocol, when serving calls invoked by the Adaptor.

2.2. UDS Interface Implementation

The UDS interface has been implemented by means of ioctl calls. In order to simplify the use of this interface, these ioctl calls can be invoked via the six primitives, introduced in section 1, implemented as library routines (send, receive, ltor, rtol) and macro-substitutions (netget, netset).

The send and receive primitives return, upon successful termination, the number of bytes transmitted or received. If an error occurs the primitives return -1; in this case diagnostic information available in the variable "_udserr" is meaningful.

The UDS interface includes facilities that allow application processes to select source addresses from which data can be received, to
allocate well-known addresses, to obtain dynamically generated port numbers, and to retrieve the signature associated with data received at a well-known address. These facilities are made available to the applications via a data structure named "net_if" and defined in [Panzieri 83]. An instance of this structure is associated with each network special file. By default, the application is provided with a dynamic port number and can receive datagrams from any network address; however, the default initialization of the "net_if" structure can be modified using netset (for example, to enable reception from a specific network address only). Similarly, a port number obtained dynamically from the driver, or the signature associated with received data, can be retrieved by calling netget.

A perhaps unusual feature of the UDS interface is that, if a datagram is received for an active application process that has not provided a sufficiently large memory area, the receive operation terminates with an exception, reported in the variable "udserr", which indicates that a "too large" datagram was received; the size of the datagram whose reception was aborted is made available in a specific member of the "net if" structure. The application can fetch this size by invoking netget, and take appropriate actions, e.g., allocating a larger area of memory and requesting the retransmission of the datagram, or implementing some flow-control mechanism.

Finally, the ltor and rtol operations disclose the mapping between "logical" and "real" network addresses as discussed in [Panzieri 83]. This mapping on the Cambridge Ring is very simple indeed, as the Ring Low interface operates on addresses identical to the "logical" addresses on the UDS interface; however a more complex mapping may be required for network-specific interfaces other than that supported by Ring Low.

2.3. Implementation of the Scatter-Gather Mechanism

The scatter-gather mechanism allows user processes to receive data into, and transmit data from, non-contiguous areas of their own memory. The basic data structure used to implement this mechanism is the buffer descriptor defined in Figure 3.

```c
struct b_desc {
    char *p_buff; /* Buffer pointer. */
    long b_length; /* Buffer length. */
};
```

Figure 3: b_desc structure.

The field "p_buff" points at a user memory area, and the field "b_length" indicates the size, in bytes, of that area. A user process may define an array of buffer descriptors and use this array as one of the input parameters to send and receive. The last element of this array must contain a NULL pointer, so that the scatter-gather mechanism can determine the number of scatter or gather buffers provided by the user process. The size of this array is limited, in our current
implementation, to a maximum of 6 elements, including the element containing the NULL pointer, because the driver stores the user-provided array of buffer descriptors in its own internal data structures. We are investigating the use of dynamic storage allocation internal to the driver in order to relax this constraint. The send and receive primitives construct the "send_buf" and receive_buf" structures of Figure 4 before invoking the ioctl that activates the driver.

```
struct send_buf { /* Used in SEND operations */
    long nbytes;     /* No. of bytes sent */
    short nbufs;    /* No. of gather buffs */
    L ADDRESS dest; /* Destination address */
    short senderr;  /* Error indication */
    struct b_desc gather[MAX_SG-1]; /* Gather buffs array */
};

struct rcv_buf { /* Used in RECEIVE operations */
    long n_bytes;    /* No. of bytes received */
    short n_bufs;   /* No. of scatter buffs */
    L ADDRESS src;  /* Source address */
    short rcverr;   /* Error indication */
    short timeout;  /* Timeout in receive */
    struct b_desc scatter[MAX_SG-1]; /* Scatter buffs array */
};
```

Figure 4: send_buf and rcv_buf structures.

The send routine assigns to "nbytes" the total number of bytes to be transmitted, to "nbufs" the total number of user buffers, and copies in "gather" the user array of buffer descriptors. As mentioned before, the data gathering mechanism determines whether the user data buffers can be transmitted in a single UDS unit, or whether multiple units are required, and loads user data into the Packet data area until either that data area is full or there is no more user data to send.

The scatter mechanism is, as may be expected, the converse of the gather mechanism. A user process invoking receive may provide an array of buffer descriptors specifying non-contiguous areas of that process' memory. The receive library routine copies these buffer descriptors into the "scatter" parameter of the "receive_buf" structure and activates the driver. The driver scatters the fragments of a datagram for that process to the buffers specified by those buffer descriptors, provided that the total amount of buffer space is sufficient.

3. Performance Measurements

In this section we discuss some performance results obtained from initial tests of our UDS interface implementation. Our aim was to measure the UDS Data Transfer Rate (DTR), as perceived by the user, and compare it with that provided by the implementation of the Ring High interface available to us. Thus, our concern was to assess the possible overheads imposed (i) by the fragmentation of large datagrams and the
use of acknowledgements within the driver, and (ii) by the scatter-gather mechanism.

The performance results discussed below were obtained by testing both the UDS interface and the Ring High interface implementations between two PDP 11/23 computers communicating via a Cambridge Ring. The tests involved a large number of runs (between 1000 and 5000 times) of a program on one cpu receiving datagrams from the Ring, and a program on the other cpu transmitting datagrams. Various datagram sizes were used, from 512 bytes to 32 K-Bytes. The standard UNIX "ftime" system call was used to obtain the timing values (note that these are only approximate, as the physical machine clock can be accurate to 1/50 of a second only).

To begin with, the maximum DTR observed using the Ring High interface was 5.54 K-Bytes/sec. The source process fragmented data objects larger than 512 bytes into 512 byte blocks, as required by that interface, and transmitted each block to a remote destination process, without caring whether the data were actually received by that process.

In order to emulate the implementation of an application-dependent protocol using the Ring High interface, we introduced a copy operation within our data source process. The DTR achieved using that interface fell to 5.07 K-Bytes/sec, indicating that the copy operation adds a further overhead of about 8%; the need for this copy operation is removed when scatter-gather facilities are available. (The DTR's above may appear rather small if one takes into account that the Ring bandwidth is 1.25 M-Bytes/sec. However, the Ring itself transmits 2 bytes of data at a time and the Ring hardware interfaces used in these tests are interrupt-driven; hence, transmission of a block of data involves considerable interrupt handling overhead.)

In testing the Ring High interface we measured a data-loss rate of up to 60%, as the data source process transmits faster than the destination process can receive. Thus, applications may introduce their own end-to-end acknowledgement to indicate, for example, that data have been successfully received by the destination process. Emulating this situation using the Ring High primitives shows a drastic degradation of the DTR to 3.61 K-Bytes/sec. This is largely because of the number of system calls required by the source process to send large data objects in fragments of 512 bytes each, and by the destination process to send acknowledgements. In our UDS interface implementation, since the successful termination of the "send" operation indicates that the transmitted data has been stored in the receiver buffers, there is no need for an application-specific end-to-end acknowledgement for this purpose.

The average UDS interface DTR when user processes use scatter-gather buffers starting at even memory boundaries is 4.76 K-Bytes/sec; this average is extrapolated from the figures in Table 1 below.
Datagram +--------------------------------+| Size | 1/2K | 1K | 2K | 4K | 8K | 16K | 32K |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>0.11</td>
<td>0.22</td>
<td>0.42</td>
<td>0.83</td>
<td>1.65</td>
<td>3.26</td>
<td>6.46</td>
</tr>
<tr>
<td>K-Byte/sec</td>
<td>4.55</td>
<td>4.55</td>
<td>4.76</td>
<td>4.82</td>
<td>4.85</td>
<td>4.91</td>
<td>4.95</td>
</tr>
</tbody>
</table>

Table 1: Average Time and DTR for Even Length Buffers.

Comparing this UDS interface DTR with that obtained using the Ring High interface with an end-to-end acknowledgement at the application level shows an improvement of about 32%. This indicates that handling the fragmentation and the acknowledgements within the driver, thus reducing the number of context switches required to transmit large datagrams, is a useful strategy in this instance.

The graph in Figure 5 shows the results obtained when the number of scatter-gather buffers varies from 1 to 5 and the datagram size varies from 512 bytes to 32K-Bytes. The continuous line refers to the UDS interface DTR obtained when the scatter-gather buffers start at even memory boundaries, and the resulting UDS datagram has even length. The dotted line indicates the UDS interface DTR when user data are gathered from and scattered to user buffers starting at odd memory boundaries, and the overall length of the transmitted and received UDS datagram has odd length.

Figure 5: Average DTR on UDS Interface.
From this graph it can be seen that the UDS interface DTR is practically constant, however an overhead of about 50% is introduced when the UDS interface deals with buffers starting at odd memory boundaries. This overhead is caused by the limitations of the "iomm" routine (or "bufmove" for machines without separate Instruction & Data space) provided within the UNIX kernel for I/O transfer between user and kernel space. This routine is slower when manipulating odd address and length buffers, as documented in [Ritchie79].

This overhead is more explicitly illustrated by the graphs in Figures 6(a) and 6(b). The graph in Figure 6(a) shows the cost of gathering 32K-Bytes of user data from 1 to 5 even length buffers starting at even memory boundaries; the graph in Figure 6(b) shows the cost of gathering (32K - 1) Bytes from 1 to 5 odd length buffers starting at odd memory boundaries.

![Graphs showing cost of gathering odd and even length buffers](image)

(a) = 32 K from Even Boundary/Length Buffers.  (b) = 32 K from Odd Boundary/Length Buffers.

Figure 6: Cost of Gathering Odd and Even Length Buffers.

These graphs indicate that the cost of gathering from even memory boundary buffers is practically negligible, and not affected by the number of user-provided gather buffers but that the cost of gathering user data from odd memory boundary buffers grows with the number of user-provided gather buffers. The limitation imposed by the implementation of the "iomm" (or "bufmove") routine, although outside the scope of our exercise, would appear to be one that could benefit from some attention.

Finally, initial tests of a UDS interface implementation for a DMA Ring hardware interface, supporting both the Ring High interface and the UDS interface, have shown that the UDS interface achieves a performance improvement of between 25% and 30% over that available using the Ring High interface with an explicit acknowledgement at the application
level. These tests were executed by the DMA station transmitting to itself around the Ring, as only one DMA interface is currently available to us; thus, comprehensive testing cannot be carried out until we have a second DMA interface.

4. Concluding Remarks

Various implementations of the UDS interface have been developed, in our Laboratory, for different communications media; our next tasks will be to assess the adequacy of this interface for communications over (i) packet-switched wide area networks and (ii) inter-networks consisting of both wide and local area networks. We also expect useful experience and results from the investigation, currently in progress, on the provision of the UDS interface to operate on Transport Service interfaces for Ethernet networks.

Our interface was designed primarily for use from within the Newcastle Connection; a modest restructuring of the Connection software is currently being carried out to use it. However, we believe that the UDS interface has more general relevance, particularly for a number of transaction-oriented applications of UNIX. As discussed in [Watson81], a datagram service is the basic transport mechanism for use with several hardware interconnection technologies, and provides adequate support for the development of application-specific protocols. In addition, issues of portability of distributed applications can be greatly aided by providing a Uniform Datagram Service which as far as possible insulates applications from the differing addressing and formatting conventions associated with different data communications technologies.

We must admit that the provision of the abstraction of very large datagrams is a rather controversial issue. Implementations of conventional datagram interfaces usually limit the size of the data object that applications can transmit and receive in a single operation to a maximum ranging between 500 and 600 bytes. However, as we anticipated in [Panzieri 83], applications can benefit from the removal of datagram size restrictions. As the implementation of the UDS interface described in this paper shows, we have been able to obtain an average performance improvement of approximately 30% in the process-to-process communications over a local area network by providing (i) the abstraction of very large datagrams, that minimize the number of context switches required to fragment and transmit such datagrams, together with (ii) scatter-gather facilities, that remove the need for copy operations at the user level. This performance improvement has been obtained without compromising the reliability of the process-to-process communications over the network.

Acknowledgements

We are indebted to our colleagues Jay Black and Lindsay Marshall for their assistance during the implementation of the UDS interface. We also wish to thank our colleagues Brian Randell and Santosh Shrivastava for their advice in the preparation of this paper. This research has been supported by the UK Science and Engineering Research Council and by the Royal Signals and Radar Establishment.
References


