Interfacing UNIX to Data Communications Networks

FABIO PANZIERI AND BRIAN RANDELL

Abstract—We propose an interface for use from within UNIX user programs for communicating over multiple and varied local and wide area networks. This interface aids the design of a distributed application program by hiding the actual communications protocols used over each network, and providing instead simple primitives for sending and receiving (possibly large) datagrams, using a simple standardized network addressing scheme based on a (host number, port number) pair.

This interface has been designed principally for use by the Newcastle Connection, a software subsystem which is used to construct the UNIX United distributed system; however, this interface may well fulfill the needs of a large class of transaction-oriented applications of UNIX. Comparisons are made with various alternative approaches to network interfacing, including the Xerox Pup communications architecture, the "Clean and Simple" approach of University College London, and the socket scheme developed for Berkeley UNIX.

Index Terms—Datagram, network protocols, networks, Newcastle Connection, UNIX

I. INTRODUCTION

In this paper we discuss the design and implementation of a convenient networking interface for use within UNIX user programs. We consider a (potentially very large) number of geographically dispersed UNIX systems, running on machines interconnected by a wide variety of different data communications facilities such as packet switching networks, local area networks, radio links, and asynchronous lines. (For convenience, in what follows we will use the term "network" for all of the different types of communication facility.) We argue that, in order to assist the construction of distributed applications on such a collection of systems and data communications networks, it is appropriate to provide designers and implementors of these applications with a uniform programming interface to those networks.

We also argue that a convenient UNIX interface to data communications networks should be characterized by a small number of primitive operations, to be invoked with a small number of parameters, and raising a small number of exceptions, similar to the interface to a peripheral device in UNIX. In other words, we suggest that the UNIX interface to a data communications network should be as simple, homogeneous, and well-defined as that to ordinary UNIX I/O devices [1]. In particular we suggest that the communications protocols implemented on a given network should be handled, at least conceptually, within the device driver providing access to that network (also called "network driver" or simply "driver" in the following), and should be invisible to the processes making use of that network. In practice, the software required to drive one or more networks might be implemented by a front-end processor; in such a case the actual driver within UNIX would in fact just implement some appropriate protocol for communication with the front-end.

In this paper we show that UNIX processes can be provided, at low cost, with network interfaces which meet the requirements mentioned above. These, we propose, are
interfaces which provide the abstraction of possibly very large datagrams, use a simple standard network addressing scheme, consisting of (host number, port number), and do not in general hide the fact of a machine being connected to several data communications networks. Our proposed interface is described in full detail in the Appendix. In summary, it consists of the following set of primitives:

1) long receive (netfd, source, pbuff, timeout): receive a datagram via network “netfd,” from port “source,” into the set of buffers “pbuff,” within time “timeout.” Normal termination of this primitive indicates that a datagram has been received and that its data and address fields are not corrupted;

2) long send (netfd, destination, pbuff): send a data-
gram via network “netfd,” to port “destination,” from the set of buffers “pbuff.” Normal termination of this primitive indicates that the input datagram has been successfully transmitted;

3) ltor (netfd, raddr, laddr): convert logical address
“laddr” to real network address “raddr” for network
“netfd”;

4) rtor (netfd, laddr, raddr): convert real network ad-
dress “raddr” to logical address “laddr” for network
“netfd”;

5) netget (netfd, pnet_ if): general parameter enquiry
function for network “netfd,” via data structure
“pnet_ if”;

6) netset (netfd, pnet_ if): parameter-setting function
for network “netfd,” via data structure “pnet_ if.”

In keeping with the existing UNIX system call interface, these are all synchronous operations, asynchronous behavior requiring explicit use of multiple processes. Note that, in essence, these can all be implemented by means of UNIX “ioctl” calls rather than as new system calls.

The adequacy or otherwise of UNIX’s interprocess communication facilities is an issue that we regard as one best treated separately from that of network interfacing. However, it should be possible to use the network primitives to transfer datagrams between ports on the same processor. (It is hoped that actual use of the network would be avoided in such circumstances.)

In line with our aim of providing a basic uniform interface, we regard the fundamental task of the networking primitives as being that of transferring a set of bytes between ports. When such transfer occurs between heterogeneous processors, various other problems must still be dealt with; in particular those due to incompatible character sets, byte ordering, floating point number representations, etc. These we regard as issues that should be handled independently of our datagram interface, either by the application programs themselves or, preferably, by what is often termed a “Presentation Layer” [2], [3]. Such matters are not considered further in the present paper.

Our interface can either be maintained by a network driver specifically constructed to support it, or can be constructed out of existing network interfaces, provided by existing drivers. This latter case involves incorporating in those drivers an “Adaptor,” a software component which maps an existing network interface into the one proposed.

This paper is structured as follows. In the next section we discuss the rationale behind our proposal. Although we believe that there is a general need for the sort of uniform network interface proposed herein, our work has been motivated by the requirement for a networking interface for use by the Newcastle Connection [4]. This is a software subsystem which we use to construct a distributed system (called “UNIX United”) out of a set of UNIX systems which are interconnected by one or more networks, possibly of various types. In Section III therefore we introduce the communication software architecture used in UNIX United systems and discuss the requirements to be met by the uniform interface we wish to define. In Section IV we examine the structuring of the communications software supporting that uniform interface, namely the structuring of network drivers, and we discuss the functionality of what we have termed Adaptors. Finally, we provide some comparisons of our approach with other relevant approaches known to us, and give some concluding remarks.

II. THE NEED FOR A UNIFORM UNIX NETWORK INTERFACE

Within the large collection of UNIX machines mentioned in the previous section, some may be directly connected to several data communications networks. It is often the case that, if a machine is connected to different networks, it is provided with different access interfaces to those networks, so that the differences between these interfaces will be reflected in the programming interface supported by the UNIX operating system running on that machine. This is because the device drivers by means of which user processes access the different networks will in general implement a network-specific communications protocol and provide a protocol-dependent programming interface. Moreover, each such interface may enforce a specific interprocessor communication model by providing a set of communications primitives and addressing conventions which are specific to the particular type of network.

Thus, for example, if a machine is connected to an X.25 packet switching network [5] and to a Cambridge Ring [6], the driver of the X.25 network might implement some Transport Service (TS) protocol [7]–[9] to support interprocess communication; this driver would then provide its user processes with primitives for establishing, maintaining and releasing process-to-process connections over the X.25 network (or a collection of interconnected X.25 networks). The Cambridge Ring driver might instead implement the Packet Protocol described in [10], provide primitives for exchanging datagrams among processes distributed on a Ring, and maintain addressing conventions quite different to those available on the packet switching network interface.

Designers and implementors of distributed applications
will in such circumstances be confronted by the problem of having to cope with a number of different programming interfaces when they require to make use of different networks. The difficulties of engineering and implementing these applications will be greatly exacerbated by the compulsary use of different primitives to accomplish logically similar functions (such as the synchronous transmission of data from one process to another), and by the different addressing schemes and exception handling which may be required by the use of different communications primitives.

One approach to this problem is to provide a network-independent transport service. Protocols providing this service have been designed and implemented so as to hide the existence of multiple networks, to the extent of providing a common addressing scheme and common data communications facilities across these networks [8], [9], [11]. However these protocols are, in general, very complex and designed to optimize the use of wide area networks, typically characterized by low bandwidth, low throughput and high error rate. On other types of network, such as high bandwidth, high throughput, low error rate local area networks, or on basic data communications media such as asynchronous lines, these protocols may not only unnecessarily affect performance. Thus, although the concept of a network-independent transport service may perhaps be attractive, in practice there are not yet communications protocols capable of providing this service cost-effectively on the large variety of data communications technologies available to distributed applications.

However we wish to construct distributed systems from small UNIX systems connected by, say, asynchronous lines, as well as from large numbers of powerful UNIX systems using sophisticated networking facilities. In the present situation we therefore are willing to allow the existence of multiple networks (but not their functional differences) to be apparent at the UNIX user program level.

The standardization process undertaken by organizations such as ISO, CCITT, ECMA, and IEEE has concentrated, so far, on the specification of network protocols, and has not yet reached the point of defining standard interfaces to networks. Indeed, the definition of these interfaces, through crucial to the designers of distributed applications, has, to the best of our knowledge, received very little attention, although suitable definitions may result eventually from current work on the specification of the Application layer of the OSI Model [2], [3].

The present lack of definition of these interfaces appears to be the result of a standardization process driven by a "bottom-up" approach; that is, by an approach which has moved from the characteristics of the specific data communications media towards the specification of communication protocols adequate for these media. However, it is also appropriate to take a "top-down" approach which, moving from the recognition of the requirements of the distributed applications that may be constructed on one (or more) networks, defines adequate programming interfaces to meet these requirements and finally specifies suitable communications protocols to support these interfaces. In fact, as we show later, it is possible, at least for the UNIX environment, to define a simple, yet effective, interface which could be supported by a variety of different protocols and could be appropriate for both local area networks and wide area networks.

In this paper we discuss a solution to the problems addressed in this section, which is based on the "top-down" approach just mentioned, yet which takes into account the realities of existing or proposed standard network protocols, and even their implementations. As indicated earlier, the development of this solution has been stimulated by our own experience in implementing the Newcastle Connection subsystem (introduced in the next section) so as to operate across multiple networks. This experience brought home to use the disadvantages of not having a uniform programming interface, and has revealed a number of inadequacies in the programming interface of the packet switching network available to us (see Section IV-E).

III. COMMUNICATIONS SOFTWARE ARCHITECTURE IN UNIX UNITED SYSTEMS

A UNIX United system is a distributed system composed of a number of UNIX (or UNIX look-alike) systems. It is functionally equivalent, both at the user and the program level, to a single-processor UNIX system. The architectures of the communication software supporting UNIX United distributed systems can be thought of as consisting of two levels of abstraction, as depicted in Fig. 1. This shows the structure of the communications software in a machine acting as a component of a UNIX United system, and physically connected to a packet switching network (PSN), two Cambridge Rings [6] and an Ethernet [12] local area network. This communication software is structured into the two levels of abstraction described below.

The Level 1 software implements the Newcastle Connection subsystem [4] and maintains the interface labelled
NC (Newcastle Connection) in Fig. 1; this interface matches that of the standard UNIX kernel, since it is characterized by the standard UNIX system calls and data objects. It is an important characteristic of UNIX that the kernel interface provides the only means by which an application process can interact with other processes, or with files or I/O devices. The Newcastle Connection intercepts all the UNIX system calls issued by application (and operating system) processes; if a call accesses an object local to the system where the process invoking that call is running, the Newcastle Connection simply passes that call to the local UNIX kernel; if the object resides on a remote system, the Newcastle Connection transforms that call into a remote system call to a server which then calls the UNIX kernel of the remote system.

All issues regarding distribution are thus handled within the Connection, in response to conventional system calls. The fact that a UNIX United system is distributed is completely hidden from the application process, as are the network protocols and addressing schemes that are used. All such processes operate as though on a single time-shared machine, and communicate with each other using standard UNIX interprocess communications facilities, which happen to involve interprocessor communications when appropriate.

UNIX system calls are essentially procedure calls—an application process is blocked whenever it makes a system call until that call is completed (successfully or not). Hence it is natural to implement the remote system call using a remote procedure call (RPC) protocol. As a result the two UNIX systems involved will be at least temporarily in an asymmetric client–server relationship, a relationship which is somewhat similar to that between, say, a process and a disk driver. The programming interface that we propose in the Appendix reflects this similarity.

The Newcastle Connection uses a protocol based on that described in [13], and whose detailed definition is given in [14], to implement remote procedure calls. For the purposes of this paper, we shall not elaborate the design of this protocol here; however, we would like to point out that this protocol serves the sole purpose of providing interprocess communication between those processes on separate systems which implement the Newcastle Connection; that is to say, it is not a general purpose interprocess communication facility which can be used by any application process running in a UNIX United distributed system. (In fact even the existence of a remote procedure call protocol is “buried” inside the Newcastle Connection software and is invisible to processes using the NC interface.) This RPC protocol is the only part of the communication architecture being described which is implemented by the Connection itself.

The implementation of the RPC protocol requires primitive operations for exchanging messages among processes distributed over a network; this requirement can be conveniently met by the abstraction of a datagram service available on that network. In order to shield the Connection layer from complexities of differing network inter-
as pointed out in Section II, at present there are no protocols that, implemented in a driver, conveniently and economically construct this abstraction out of any sort of data communications technology. We therefore regard it as necessary to allow internetworking issues to be handled within the Level 1 remote procedure call protocol, as well as within one or more drivers.

IV. NETWORK DRIVERS

As introduced in the previous section, the Level 0 drivers each provide access to one or more data communications networks and uniformly support the K interface. In order to maintain this interface, each driver performs two conceptually distinct functions. Firstly, each driver implements some “network-specific” functions consisting of network-specific communications protocols (possibly associated with network-specific routing and flow control policies) and maintains some programming interface which also will be, in general, network-specific. Secondly, each driver performs a set of “network-independent” functions so as to hide the network-specific interface and to map this onto the K interface.

The distinction between these two functions may not be apparent in the internal structure of the driver if it is constructed specifically to support the K interface; however, for the purposes of our discussion we shall maintain this distinction and we shall make the following two assumptions.

1) Network-specific and network-independent functions are implemented separately in a driver by two distinct components, namely a Network-Specific (NS) Driver and an Adaptor. Thus, the structure of a driver can be thought of as depicted in Fig. 2.

2) Although distinct Level 0 drivers may incorporate different NS Drivers which implement different NS Interfaces, these interfaces have the common feature of providing primitive operations for addressing either other hosts or other processes distributed over a network, and for transmitting data to and receiving data from those hosts or processes.

As far as the second assumption is concerned, in the case where the NS Driver provides only host-to-host communication the Level 1 abstraction of process-to-process communication would have to be implemented by the Adaptor of the Level 0 driver. For example, if a Level 0 driver provides access to an X.25 packet switching network it may incorporate an X.25 Driver whose NS Interface supports virtual circuits between pairs of host computers. The Adaptor would then use this internal interface to implement some form of virtual circuit multiplexing to allow communication between processes on the host computers at the end points of a virtual circuit. Protocols providing the required process-to-process communication abstraction are well known in the literature [9]-[10], [15] and widely implemented (e.g., PSSNET, SERCNET, Cambridge Ring, Arpanet, Xerox PARC). In what follows we will, for the sake of simplicity, assume that each NS Interface defines process-to-process communications.

In the rest of this paper we shall not elaborate further on the network-specific functions performed by NS Drivers; rather, we shall concentrate our discussion on the design and implementation of Adaptors.

A. Adaptors

The network-independent functions of an Adaptor consist of 1) providing the abstraction of a uniform datagram service (UDS), described below, and 2) providing the primitives required for the K interface. We should like to point out that we use the term “abstraction” of a datagram service to indicate that this service must only appear as such to the Level 1 software; the actual implementation of this service is of no concern to that software.

The UDS can be summarized as follows. Each Adaptor provides its user processes with the same set of primitive operations and process addressing conventions to transmit and receive a data unit of fixed maximum size, i.e., a datagram, over a network; each datagram is transmitted on a network independently of all other datagrams sent on that network.

As argued elsewhere [13], [16], [17], distributed applications can be constructed by utilizing an adequately reliable datagram service. These applications can always implement their own reliability mechanisms, if necessary, and in fact will typically have to do so, in order to cope with problems of host reliability and availability. Thus, the K interface should be supported “sufficiently reliably” (a characteristic that we do not propose to quantify) for users to be able to assume that 1) if the “send” operation terminates normally, the datagram has been transmitted with a high probability of success, and that 2) if the “receive” operation terminates normally, the data and address fields in the datagram that has been received have not been corrupted. That is, users should be able to assume that such faults are either masked or are reported by appropriate exceptions on the K interface (see Section IV-F). Moreover, it should be of no concern whether the K interface achieves such reliability because of the intrinsic reliability of the communications hardware, or by fault tolerance strategies incorporated within the NS Driver or its Adaptor. The fault tolerance strategies which might be appropriate include, for example, acknowledgments, retries, etc.

The K interface maintained by an Adaptor consists of the six primitives listed in the Introduction, and defined in the Appendix. These primitives are implemented by the drivers of each particular network; thus, in order to invoke them, an application process must first activate the driver which implements them. The following mechanism is available in UNIX for this purpose.
As mentioned before, each network to which a UNIX system is directly connected can be represented by at least one special file [1], normally held in the directory "/dev" of that system. A special file is characterized by a (major device number; minor device number) pair and can be acted upon using the same operations as for ordinary disk files. When a network special file is opened by an application process invoking the UNIX "open(...)" call, UNIX uses the "major device number" of that special file to select the appropriate driver. The "minor device number" is passed to the driver for its own internal use. For example, the Driver B in Fig. 1 might use the "minor device number" to select one of the two Rings to which is provides access, in the same way that a driver of a number of disk units of the same type uses the "minor device number" to select one particular unit.

The file descriptor returned by the "open(...)" primitive to the calling process can then be used by that process to perform any further operations on that special file, and hence on the network; the appropriate routines implemented by the network driver will be executed to perform the operations requested by the application process. This file descriptor, and the possible network resources that the driver may have associated with it, are released when the UNIX "close(...)" operation is applied by the application process to that file descriptor.

To conclude this section, we would like to reiterate that the interface proposed in this paper satisfies our requirement to support the Newcastle Connection subsystem and, we believe, the requirements of a large class of transaction-oriented applications. However, applications other than these may also require the ability to use the primitive operations available on the NS Interface underneath an Adaptor (e.g., remote terminal access on X.25 networks). Thus, the software supporting the K interface may well be organized as illustrated in Fig. 3.

Fig. 3 shows an Adaptor which "extends" an NS interface with the primitives we propose, in order to support the Newcastle Connection subsystem, without affecting the programming interface required by an application using that NS Interface.

B. Choosing the Size of the Data Field in the Datagram

The maximum size of the data object to be made available on the K interface could be fixed according to any of the following three different approaches.

The first approach is to base the maximum size of the data object on that which can be handled on an NS Interface. As distinct NS Interfaces may support the manipulation of data object of different maximum sizes, a decision would have to be taken as to whether to support on the K interface either 1) a data object whose maximum size varies from driver to driver, or 2) a data object whose maximum size is fixed to the smallest among those provided by each driver. Both these alternatives are undesirable. The former would require the Level 1 software to implement different fragmentation strategies depending on the network it accesses; the latter would force the Level 1 software to fragment an unnecessarily large number of data objects.

The second approach, implemented in the Pup architecture [18] and in the Catenet scheme [19], is based on observation of average network traffic. Most of the network traffic may consist of individual disk pages of 512 bytes each, say, so the size of the data field is based on this figure. Since the data field in the datagram has to accommodate higher level protocol overhead, it is necessary to allow additional bytes for such purposes. In Pup, for example, the datagram data field has a size of 532 bytes, allowing 20 bytes for higher level protocol overheads; in Catenet it has a size of 576 bytes, allowing 64 bytes. Adaptors implementing the same abstraction as Pup and Catenet must provide fragmentation and reassembly of the datagram data field when this is to be transmitted on networks that manipulate smaller size data objects. Additional fragmentation and reassembly must be performed by the higher level software when objects of size greater than 512 bytes have to be transmitted.

The third approach, which we prefer and have adopted, is based on the following arguments. It is convenient that the maximum size of the data unit maintained on the K interface be very large in order to allow application processes using that interface to transmit large quantities of data in (what appears to be) a single operation. Should the NS Interface supporting an Adaptor not provide facilities for the transmission of very large datagrams, the Adaptor will be required to fragment such datagrams in order to transmit them, and to reassemble the received fragments, in order to deliver the datagram to the destination process. Since fragmentation and reassembly are to be performed at some level of the communication architecture, they may as well be implemented once and for all in one level only. In addition, since we believe it appropriate for the network interface to be as similar as possible to that of an ordinary I/O device, such as a disk, say, the network interface should only impose restrictions on the size of the data object that are similar to those imposed by the interface to an ordinary I/O device. Based on the size restrictions in various existing versions of UNIX, we have decided that an Adaptor should use a 32 bit field to represent that size and so allow for a data field of 2048 Mbytes.

Clearly, processes will not normally contain memory regions of this size. Nevertheless an Adaptor which is required to transmit a very large data object must do so in a fashion which is transparent to the transmitting process.
The Adaptor is ideally situated to exploit network-specific fragmentation and transmission policies. On reception, it is the responsibility of the receiving process to provide the Adaptor with a sufficiently large area. (An attempt to receive a datagram which is larger than the area of memory provided by the receiving process will result in the Adaptor returning an exception to the receiving process.)

Perhaps the most important argument for this third approach concerns the use of typical wide area networks. The task of transmitting a large quantity of data over such a network with adequate performance and reliability requires a sophisticated protocol, not just the use of a sequence of limited-size datagrams. If the K interface supports "send" and "receive" of large data objects there will be scope for the use of an appropriate NS driver. Indeed one could envisage the sort of Adaptor illustrated in Fig. 4, which uses a simple NS Datagram protocol for small objects, but a more complex transport protocol [8], [9], [11], for large objects, hiding such details from the user programs.

Problems related to having different naming and addressing conventions maintained on the NS Datagram Interface and the NS-BSP Interface can still be solved within the Adaptor. For example, the Adaptor of Fig. 4 may implement appropriate functions to map both NS Datagram Interface addresses and NS-BSP Interface high-level names into K addresses. (Issues related to address mapping are discussed further in the next section.)

When applications use very large data objects, it is desirable to provide them with means for avoiding unnecessary copying of these objects from one place to another within memory. To this end, Adaptors provide "send" and "receive" operations that gather output data from, and scatter input data to, a specified list of noncontiguous areas of the user process memory. User processes implementing application-specific protocols, for example, can pass to the Adaptors pointers to the relevant areas of memory containing data to be transmitted, or where received data is to be placed. Thus, overheads imposed by the copy operations which would be required if scatter/gather facilities were not available are removed; in addition, the use of sophisticated DMA devices can be exploited when these devices are available.

We would like to emphasize that an Adaptor is not expected to perform internetworking functions; hence, conventional internetworking problems, such as those due to the differences in speed and maximum packet size of different networks [20], should not arise in the construction of an Adaptor. Rather, in our architecture, conventional internetworking functions could be implemented by the NS Driver supporting an Adaptor (e.g., an NS Driver implementing Pup). In addition, the Newcastle Connection software implements some internetworking functions specific to the Connection itself.

### C. Network Addressing

An "address" in a data communications network indicates the location of a resource on that network [21] (e.g., the address of a Line Printer Server, that of a Name Server, etc.); an "address space" is the collection of addresses associated with the set of resources available on a network. The abstraction of an address space is maintained by the NS Interface which provides access to a network. Examples of address spaces are: the set of "transport service addresses" on a Transport Service interface [9], the set of "network addresses" on the Pup interface [18], the set of pairs (station_number, port_number) on the Cambridge Ring Packet Level Protocol [10] interface. Although an address space may encompass several actual networks, it can be considered to be a single network, as this is the abstraction provided by the NS Interface maintaining that address space. Thus we very deliberately do not define (or adopt) any explicit internetwork addressing scheme, in order that internetworking should be possible above or below the K interface.

Different NS interfaces are likely to use different address formats. For example, an address on a Transport Service interface consists of a string of characters; on the Cambridge Ring interface it is represented by two integers. On the other hand, an important characteristic of the K interface is that it provides a single uniform address format. Therefore each Adaptor maintains on K a set of "logical" addresses that it maps into the set of "real" ones on the NS Interface beneath it. The addresses provided by the K interface are defined as follows:

```c
typedef struct _address{
    unsigned long host_number;
    unsigned long port_number;
} L_ADDRESS;
```

The data type `L_ADDRESS` consists of two 32 bit unsigned integer fields; the field "host_number" identifies a host computer on a network, and the field "port_number" identifies a subaddress within that host. (Since the fields comprising a logical address are purely numerical, and carry no encoded information, we regard 32 bits as being more than sufficient.)
The logical addresses maintained by an Adaptor on the $K$ interface are meaningful only to that Adaptor and to the process using (the Driver containing) that Adaptor. That is, Adaptors in two distinct Drivers may well maintain two distinct logical address spaces characterized by identical \langle host number, port number \rangle pairs. No confusion can be generated at the application software level, since those two address spaces are separate and distinct on $K$; the two networks to which they are associated will be represented on $K$ by (at least two) different special files. Furthermore, Adaptors for the same network but different hosts are at liberty to maintain different logical to real address mappings.

This particular format of the logical addresses maintained on the $K$ interface has been chosen mainly for its simplicity; in essence, it allows the application software to implement conveniently the binding and resolution of resource names into the (logical) addresses of these resources. The Newcastle Connection subsystem, for example, binds some special file names, corresponding to UNIX systems distributed on a network, to the network addresses of the machines on which these UNIX systems are running. When necessary, the Connection locates a remote UNIX system by resolving the name of the special file corresponding to it into a network address.

The data type $\text{L\_ADDRESS}$ is used to specify the destination address of a datagram. For example, the primitive implementing transmission of a datagram within an address space uses this data type as indicated below (the semantics of this primitive are discussed in the Appendix):

```c
long send(netfd, destination, pbuff)
    int netfd;
    L\_ADDRESS *destination;
    struct b\_desc *pbuff;
```

The parameter \"netfd\" is a file descriptor returned by a successful (standard UNIX) \"open(\ldots)\" of a special file associated with the physical network; \"pbuff\" is the address of an array of descriptors of buffers whose contents are to be sent to the (logical) network address pointed at by \"destination.\" Each buffer descriptor consists of the base address of the transmit buffer and the length of that buffer.

The construction of logical address spaces on the $K$ interface is illustrated by the following three examples. The Adaptor in Driver $A$ of Fig. 1 may be implemented on an NS Interface supported by a \"Yellow Book\" [9] Transport Service Driver. A \"real\" address on that NS Interface consists of a string of characters, such as \"1500100/ A\_SERVICE.\" This string of characters is composed of two distinct parts separated by the character \"\/.\" The set of characters to the left of the \"/\" indicates the DTE address of a host computer on the network accessible by means of that NS interface, the remaining part indicates the server process to be used at that host address in order to provide the service named \"A\_SERVICE.\"

The Adaptor implemented on that NS Interface can provide the abstraction of a logical address space on $K$ by mapping the host numbers on $K$ into the DTE addresses on the NS Interface, and the port numbers on $K$ into the service names (and vice versa, when required). The efficiency with which this mapping function can be performed is largely dependent on the memory resources available to the Adaptor. Thus, for example, if transport service addresses were maintained on disk files, the Adaptor may wish to cache the most frequently used transport service addresses and perform disk accesses only when necessary.

A much simpler mapping function can be implemented by the Adaptor in Driver $B$ of Fig. 1, which we presume is constructed on the Ring Interface supported by the implementation of the Packet Level Protocol [10]. This NS interface maintains two identical real address spaces, one for Ring 1 and one for Ring 2, consisting of \langle station number, port number \rangle pairs. On each real address space, the \"station number\" will be in the range [0, \ldots, 255], the \"port number\" in the range [0, \ldots, 65535]. Clearly, in this case, the mappings that are used to generate the \langle host number, port number \rangle pair on $K$ can be extremely simple.

As a third example, suppose that addresses are managed by a Name Server. Then, the mapping between logical and real addresses may well be performed by the Name Server itself, or perhaps by an Adaptor at that Server, using whatever technique is appropriate. Thus, Adaptors incorporated in the network drivers could implement a request/response interaction with (the Adaptor at) the Name Server to obtain, for example, the real addresses corresponding to logical ones, and vice versa. Yet again, the number of interactions that an Adaptor would have to perform with the Name Server could be minimized by allowing the Adaptor to cache the most frequently used addresses, and interact with the Name Server only when necessary.

Port numbers on the $K$ interface provide a means of identifying the particular process, within a given host, which sent (or is to receive) a datagram. Such port numbers will in fact correspond to some appropriate mechanism provided within the UNIX kernel, the NS Driver and/or the Adaptor. In the first two examples above the port numbers correspond directly to a Transport Service address, and a Cambridge Ring port, respectively. Other possibilities might include a Berkeley UNIX socket, or the name of a system resource such as a port in the Accent system [22].

The $K$ interface provides a process which receives a datagram with a means of identifying the process which sent it. If the NS Interface supporting an Adaptor is such that it provides this facility, e.g., a \"Yellow Book\" Transport Service interface, then that Adaptor simply has to map the source addresses returned by its NS Interface into $K$ port numbers. In contrast, if the NS interface does not provide this facility, e.g., the Cambridge Ring Packet Level interface, the Adaptor transmitting a datagram can include in that datagram some identifier of the source process, such
as a Cambridge Ring port number. This identifier can be dealt with by the Adaptor at the receiving end as the source address of the received datagram, i.e., it can be passed to the application software as a K port number, which can be used to transmit datagrams back to that address. (Note that this requirement conforms to the IEEE Project 802 proposal for the standard Logical Link Control (LLC) protocol for local area networks [23].)

It is worth mentioning that Adaptors, because of the semantics of the datagram service they implement, do not have to guarantee that the correspondences between processes and K port numbers remain fixed. That is, should a machine crash occur, the port numbers previously assigned to the processes on the crashed machine may be reassigned to different processes after the crash; it will be the responsibility of the application software to implement, if necessary, appropriate fault-tolerance measures to cope with these undesirable events.

A further notion associated with an address space is that of “well-known addresses” available on that address space. It may be desirable for a distributed application to define one or more addresses, within an address space, which are statically assigned to specific servers. To this end, an address space can incorporate a set of “well-known addresses” which, once assigned to a resource, are not reused. Examples of well-known addresses in existing distributed systems are the addresses of the Spawner processes in a UNIX United system, the “well-known sockets” of the Listener processes [24] and of the Routing Information processes in the Xerox communications architecture [18]. Adaptors can provide the application software with the ability to define well-known addresses by reserving a set of port numbers (e.g., the set of port numbers in the range 0, · · · , 1023) which will be statically assigned to application processes only upon explicit request of these processes (and subject to the availability of these port numbers).

Note that the mapping between logical and real addresses performed by an Adaptor will be, in general, transparent to the application software. However, it is necessary that Adaptors provide facilities both for disclosing the mapping they implement, and for altering it. Applications may occasionally require knowledge of the real addresses maintained on the NS Interfaces; in addition, system administration utilities will be required to modify and extend the mapping table of an address space. However our expectation would be that user programs would be designed so as to minimise and localize the use of such network-specific facilities, and would whenever possible rely on logical addresses.

To meet the requirement for disclosing and altering the mapping, we propose that the K interface provide two primitives “ltor(· · · )” (logical to real) and “rtol(· · · )” (real to logical), described in the Appendix. The former takes as input a logical address in an address space, consults the associated mapping table, and returns the corresponding real address. Conversely, “rtol(· · · )” implements the opposite function: given a real address it consults the mapping table and provides a logical address to be used on K. The construction and updating of these mapping tables is a separate issue, specific to each Adaptor and beyond the scope of this paper.

D. Signed Datagram

A distributed application may require the use of “signed” messages, particularly in order to communicate with specific servers at well-known addresses. A signed message contains some (trustworthy) information as to the identity of the process originating the message. Servers may require signed messages from their clients in order to implement access control functions based on the client’s identity. In a UNIX United system, for example, messages directed to the Spawner processes must be signed. An appropriate UNIX signature consists of the real and effective <user_id, group_id> pairs associated by the UNIX kernel with the process.

Since server processes have no control over the generation of signatures on client messages, it is critical that these signatures cannot be forged by the application software. In order to meet this requirement, some actions have to be taken below the kernel interface, i.e., in the Adaptors, where the application software has no access.

Thus, Adaptors must recognize the well-known addresses in the address space they maintain. When transmitting a message to one of these addresses, the Adaptor responsible for the transmission signs the message with the signature of the source process; when receiving a message for a well-known address the Adaptor provides the destination process with the signature. This approach removes from the application software any responsibility for signing messages and also prevents any attempts at forgery. Although this approach imposes overhead on messages to well-known addresses even when this may not be strictly necessary, it has the advantage of being very general, as it allows the implementation of Adaptors which are completely application-independent.

E. Adherence to UNIX I/O Semantics

As far as possible, an adequate programmable network interface for UNIX processes must behave, in all respects, like the interface to any other UNIX I/O device. Most critical is the requirement that any UNIX I/O device driver must maintain the semantics of all the system calls provided by the UNIX kernel. This is most easily done by arranging that the I/O device interface is provided by software in the form of a device driver in the kernel. What in fact is at stake is the proper functioning of the “fork” and “exec” system calls in programs making use of a network.

“Exec( )” calls allow the invoking process to execute a named program; an “exec( )” call, when invoked, overlays the code and data of the calling process with the named program, executes that program and exits, if successful, without returning control to the calling process image. (Control is returned to the calling process image only if the “exec( )” fails.) The significant point is that files and pipes which were opened before invoking an “exec( )” remain open across that “exec( )”; signals
which were set to be ignored before an "exec()" remain ignored across it. The "fork()" system call splits the invoking process into two independent processes with identical process images, i.e., register values, open files, etc.; the only difference between these two processes consists of the process identifier associated by the UNIX kernel with each process.

For a network, or indeed any form of I/O device, to be generally usable from a variety of application programs, it is essential that the semantics of "exec()" and "fork()" be adhered to by the driver—indeed this is standard practice with UNIX I/O devices. The contrast can be seen in examining the differences between the UNIX software provided for the Swedish TELEPAK X25 network [25] and that developed for the UK SERCNET/PSS X25 networks [26]. The former adheres to the UNIX system call semantics, the latter does not. Thus although the latter provides X25 PAD and File Transfer protocols, its programmable interface is implemented largely as a subroutine library, rather than as a straightforward I/O driver. This approach is perfectly adequate for the original requirement of allowing terminal users to access other computers and request file transfers, but with hindsight has proved quite inappropriate for more general use of these facilities from within user-written UNIX application programs.

F. Exception Reporting

Based on the approach to fault-tolerance and system reliability described in [27], we view the network interface as providing normal and defined exceptional responses to each primitive it supports. For example, the normal response in the case of the receive operation would usually be taken by the caller as indicating that a datagram had arrived and was error free. In other words, any error detection needed is assumed to be responsibility of the network-specific driver and/or the Adaptor, rather than the application program. The purpose of the exceptions is to allow the application program to incorporate error recovery (possibly supplementing fault tolerance mechanisms that are hidden within the driver).

An exceptional response is the means by which the Adaptor indicates that a fault has occurred. We propose that there be just one possible exception, that one of the tasks of the Adaptor is, where necessary, to map any network-specific exceptions into the standard exception.

The proposed set of standard responses consists of

1) a normal response indicating that an operation completed successfully.
2) An exceptional response indicating that a fault occurred and the operation should not be presumed to have been completed successfully. This response can be used to map exceptional responses such as, for example, the "Not-Transmitted" response which may be obtained from the Cambridge Ring Packet Level interface [10], or the "Excessive-Collision-Error" response from an Ethernet interface [28], or the expiry of a timeout that controls the termination of some network-specific operations (e.g., the timing out of a connection request on an X.25 network).

In some circumstances it will however be necessary for the application software to be aware of the particular network-specific exception that has occurred. We therefore propose that such information be coded by the Adaptors in a variable termed "udsert" which, similarly to the UNIX variable "errno", can be examined by the application software when an exceptional response occurs. This permits appropriate monitoring and diagnostic routines to be implemented by this software, if required.

V. ALTERNATIVE SCHEMES

Work on network protocol standardization has, as we have indicated above, been more concerned with defining the communications protocols to be implemented on different data communications media, rather than with defining standard programming interfaces to these media. However, this latter issue has been faced by the designers of various manufacturer-specific networking systems. In DECnet, for example, various types of programming interfaces are defined, of which that for network-transparent task-to-task communications [28] perhaps comes closest to what we have proposed here for UNIX. This interface allows its user processes, which may be running under different (proprietary) operating systems and may be written in different programming languages, to transfer data over DECnet in a "device-independent" fashion, i.e., regardless of the actual implementation of the data transfer.

However, this interface enforces a communications model based on the establishment, maintenance and release of process-to-process connections. That is, in order to exchange data, two processes using this interface have first to establish a communication link (i.e., a connection) between them by explicitly accomplishing a handshaking sequence; when this handshake terminates successfully the data exchange can be carried out; finally, the two processes can release the link. Although this communications model may be adequate to implement some applications, such as file transfer, where the overheads introduced by the establishment and release of process-to-process connections may be negligible compared to that imposed by the actual file transfer, it is not so appropriate for transaction-oriented applications such as the Newcastle Connection which simply require the ability to send messages to and receive messages from perhaps a large number of different processes distributed on a network.

A number of different approaches to the provision of a UNIX interface to data communications networks have been developed by other groups, for example [29]–[31]. In addition, University College London (UCL) has developed a UNIX program interface to networks that, both in its objectives and in its engineering, is very similar to the uniform interface we propose. The UCL interface, (devisedly) termed "Clean and Simple" [32], provides UNIX programs with a uniform access mechanism to various networks, including the British PSS and SERCNET and the DARPA Internet, and with the ability of transfer-
ring unbounded streams of bytes over these networks. The software supporting this interface has been constructed on top of existing programming interfaces provided by these networks. In essence, the most substantial difference between "Clean and Simple" and the interface we propose is that the former provides a connection-oriented interface while the latter provides a datagram interface; thus, yet again, although "Clean and Simple" may be appropriate to support applications such as, for example, remote terminal access, it is not so appropriate for a transaction-oriented application such as the Newcastle Connection, for the reasons mentioned above.

The other approaches known to us which are most relevant to our proposals are one emanating from Berkeley and, although not directly related to UNIX, one developed at Xerox. In fact, our approach to the definition of a uniform network interface for UNIX has been notably influenced by the study of the Xerox Pup Internet Architecture [18], and of the networking facilities of the Berkeley 4.2BSD version of the UNIX system [33].

The Pup architecture is based on the Internet Datagram Protocol (IDP) [15]. This protocol implements a process-to-process datagram service between processes distributed over one or more (possibly different) interconnected networks, and maintains a uniform access interface to these networks.

The IDP interface can be thought of as being characterized by 1) primitive operations for transmitting and receiving "internet packets" (of 532 byte maximum length, as mentioned earlier) across the network, and by 2) a uniform datagram addressing convention, which applies to any network to which a machine is connected. In addition, the IDP interface maintains the abstraction of "well-known addresses," which can be statically allocated to specific servers.

The main differences between the IDP interface and our K interface reside in the choice of the maximum size of the datagram and in the structure of the network addresses where datagrams can be transmitted and received. As discussed in Section IV-B, we have chosen to allow datagrams to be effectively unbounded in length. Our addressing scheme, for the reasons discussed in Section IV-C, deliberately avoids any explicit involvement with internetworking. Needless to say, these differences in no way prevent the use, within our communications architecture, of a driver implementing the IDP protocol; providing an Adaptor for such a driver would, we believe, be a simple exercise.

The Berkeley 4.2BSD system supports a rather sophisticated interface for interprocess communication, both within a single 4.2BSD system and between systems distributed over data communication networks. This interface maintains the abstraction of several distinct domains (or "address families"). A domain represents a communications environment characterized by specific communications facilities, e.g., the UNIX environment characterized by both pipe-like and message-based communications facilities, or the ARPA internet environments characterized by virtual circuit facilities. These facilities are made available to user processes via a standard set of fourteen communications primitives. This standard set of primitives can be used for communicating on each domain.

Processes within a domain can communicate by sending and receiving messages between communications endpoints, termed "sockets." A socket is a typed system object used by user processes willing to communicate in a domain. The socket type (e.g., "datagram," "stream," "sequenced") is selected by the user process and determines the semantics of the communications that the user process will use; in the UNIX domain, for example, a "stream" socket will provide pipe-like communication, in the internet domain it will provide transport service-like communication instead.

At the user level, the abstraction of a domain may appear to be similar to that of a virtual address space maintained by our K interface. Berkeley 4.2BSD domains can be accessed by invoking the "socket(…)" primitive which returns an integer descriptor to be used for any further communication. This primitive takes as input parameters an address format descriptor, an indication of the type of communication required (e.g., datagram, stream), and an indication of the communication protocol to be used. An address space on the K interface can be accessed by applying a standard UNIX "open(…)" call to a special file representing a network, as mentioned before; the file descriptor returned by that call is for use in any further operation on that network. However, the abstraction of a domain does not hide the nature of the underlying communications environment, as is the case with our abstraction of an address space; rather, in order to communicate within a domain, processes may require knowledge of some of the characteristics of the communication environment that the domain represents—for example, in order to select an appropriate socket type to use.

In summary, the UDS interface is much simpler than the 4.2BSD interface, in large part because it deliberately provides only a small subset of the functionality provided by the 4.2BSD sockets. However, in so doing, it avoids requiring processes to be "domain knowledgeable," and makes feasible implementations on much more moderately sized machines than are typically required for 4.2BSD.

Needless to say, our K interface could well coexist with the 4.2BSD interface as, from a functional point of view, our K interface may be thought of as a subset of the 4.2BSD one. An Adaptor for the 4.2BSD interface would be required, though, in order to meet the specifications of the K interface, but this Adaptor would be very simple indeed. Its principal task would be to map the format of the network addresses as maintained by the 4.2BSD system into the required format on the K interface, and also to deal with the possibility of very large datagrams.

VI. CONCLUSIONS

It is perhaps worth summarizing the basic structuring ideas which have motivated this proposal. These are as follows.

1) Interprocess communication and networking are sep-
arable issues; hence, our interface is patterned after the standard UNIX I/O interface rather than some existing or new interprocess communications scheme.

2) The actual networking protocols used should not be evident at the UNIX user programming interface—this leads to the proposal that the user program should be presented with the abstraction of a datagram which can be essentially any size, from zero bytes upward.

3) As far as possible, user programs should be shielded from the fact that network addresses can take various forms.

4) Internetwork addressing should not be part of the network interface—we believe that it should be possible for internetworking to be implemented either at user program or at network driver level, or indeed both.

The UDS interface can of course be supported by network drivers specifically constructed for this purpose. However, we have also shown a structuring approach, based on the incorporation of Adaptors in existing network drivers, thus minimizing the amount of code to be written in order to support the UDS interface.

The present proposal has been motivated here by reference to the Newcastle Connection; however, we believe it has general relevance. We have already shown how it could be supported by a wide variety of underlying network protocols. We would also argue that it provides a suitable basis for the programming of a variety of transaction-oriented network applications based on UNIX. This is because, in effect, all that the Connection requires is a means of transferring bytes from one port to another. For example, we believe that our interface will be suitable for use by software which transmits bit maps to and from graphics terminals, provided of course that underlying device drivers and local area network provide sufficient bandwidth.

Various implementations of the UDS interface have been developed in our Laboratory; in particular, for Cambridge Ring interrupt driven interfaces, for Cambridge Ring DMA interfaces, for Ethernet, and for RS232 interfaces to asynchronous lines. In addition, a UDS interface for use on an X.25 network, and a UDS Adaptor for 4.2BSD sockets, are currently being implemented.

As shown in [34], initial testing has produced very encouraging results. For example, Adaptors have been incorporated in one of the UNIX United systems in our Laboratory, in fact within the network drivers of two UNIX systems running on two PDP 11/23 computers interconnected by a Cambridge Ring. The original Ring driver allowed user processes to transmit and receive data objects of 512 bytes maximum size; the same driver incorporating an Adaptor allows the exchange of data objects of 2 Gbytes maximum size. The data transfer rate between user processes exchanging large data objects has improved by about 30 percent. An even more notable improvement has been obtained by incorporating Adaptors in the network drivers of two PERQ computers interconnected by an Ethernet network. The communication speed between these two machines has in fact improved by over 100 percent. These improvements have been obtained by 1) handling the fragmentation and reassembly of large data objects within the Adaptors (i.e., in the UNIX kernel) rather than at the user process level, thus reducing the number of context switches between user and kernel space that were previously required, and by 2) using the "scatter-gather" facilities in order to remove the need for copy operations from the application level.

The present interface definition, it must be admitted, is more a product of hindsight than foresight. It has arisen out of work at Newcastle on extending UNIX United to work over multiple and varied networks, and from our and other groups' experience of porting Release 1.0 of the Connection onto different local networks. Moreover, as we have tried to indicate above, it has greatly benefited from the understanding that we believe we have gained from work elsewhere on interfacing UNIX to networks. The proposal will be used to guide a planned modest restructuring of the relevant Newcastle Connection software.

APPENDIX

THE PROPOSED NETWORK INTERFACE

This Appendix describes the proposed network interface. This description is provided in form of UNIX Programmer's Manual pages for the interface. It is recommended that the Section entitled NET_IF(4) be read first.

The data type "unsigned long" in this Appendix is intended to denote a 32 bit quantity.

**LTOR(2)**

**Name**
ltor, rtol - convert addresses

**Synopsis**

```
#include <udsio.h>

ltor (netfd, raddr, laddr)
int netfd;
R__ADDRESS *raddr;
L__ADDRESS *laddr;

rtol (netfd, laddr, raddr)
int netfd;
L__ADDRESS *laddr;
R__ADDRESS *raddr;
```

**Description**

These primitives disclose the mapping maintained by the Adaptors between logical and real addresses and vice-versa.

Ltor discloses the mapping between the logical address and its corresponding real one; it takes as input the logical address "laddr" and loads its corresponding real address into the output parameter "raddr."

Rtol discloses the mapping between a real address and its corresponding logical one. It takes in input the real address "raddr" and loads its corresponding logical one into the output variable "laddr."

**See Also**

udsio(5)
Diagnostics
These primitives return 0, if successful; -1 if the address whose mapping is to be disclosed cannot be found.

NETGET(2)

Name
netget, netset - enquire, instruct network driver

Synopsis
netget (netfd, pnet_if)
    int netfd;
    struct net_if *pnet_if;
netset (netfd, pnet_if)
    int netfd;
    struct net_if *pnet_if;

Description
The parameter "netfd" is the file descriptor obtained by a successful open(2) of a special file associated with a network; "source" is an output parameter which contains the source address of the received data, when this primitive terminates successfully. The parameter "pbuff" points to an array of buffer descriptors (see udso(5)). Data are received from the network as a single "datagram" and scattered to the various buffers whose address are provided in the array of buffer descriptors. Note that the receive(2) primitive does not maintain the byte boundaries of the buffers as they were before transmission. Finally, the parameter "timeout" indicates in clock ticks how long the process invoking the receive(2) primitive is willing to wait for data to be received from the network.

If the "timeout" parameter is set to the value TIMEOUT, defined in udso.h (see udso(5)), no timeout is associated with this invocation of receive(2). If a negative value is given to this parameter, the timeout is maintained as set before the receive(2) operation was invoked. If the timeout expires while a receive(2) operation is in progress, this operation terminates with value -1 and the variable "udcerr" is set to indicate that this particular event occurred.

This primitive returns a long integer indicating the total number of bytes actually received. The receiver bytes have been scattered in the buffers pointed at by the buffer descriptors in "pbuff."

See Also
open(2), net_if(4), udso(5)

Diagnostics
These primitives return 0 if successful; otherwise -1. The external variable "udcerr" is set if an error occurs.

RECEIVE (2)

Name
receive - receive a datagram

Synopsis
include <udso.h>
long receive (netfd, source, pbuff, timeout)
    int netfd;
    L_ADDRESS *source;
    struct b_desc *pbuff;
    short timeout;

Description
This primitive is used to receive data from the network; at most 2 Gbytes of data can be received. The parameter "netfd" is a file descriptor obtained by a successful open(2) of a special file associated with a network; "source" is an output parameter which contains the source address of the received data, when this primitive terminates successfully. The parameter "pbuff" points to an array of buffer descriptors (see udso(5)). Data are received from the network as a single "datagram" and scattered to the various buffers whose address are provided in the array of buffer descriptors. Note that the receive(2) primitive does not maintain the byte boundaries of the buffers as they were before transmission. Finally, the parameter "timeout" indicates in clock ticks how long the process invoking the receive(2) primitive is willing to wait for data to be received from the network.

If the "timeout" parameter is set to the value TIMEOUT, defined in udso.h (see udso(5)), no timeout is associated with this invocation of receive(2). If a negative value is given to this parameter, the timeout is maintained as set before the receive(2) operation was invoked. If the timeout expires while a receive(2) operation is in progress, this operation terminates with value -1 and the variable "udcerr" is set to indicate that this particular event occurred.

This primitive returns a long integer indicating the total number of bytes actually received. The receiver bytes have been scattered in the buffers pointed at by the buffer descriptors in "pbuff."

See Also
open(2), net_if(4), udso(5)

Diagnostics
The return value -1 indicates that an error has occurred. The content of the buffers pointed at by the descriptors in "pbuff" and of the output parameter "source" are to be ignored; the external variable "udcerr" contains an indication of the type of error.

SEND(2)

Name
send - transmit a datagram

Synopsis
include <udso.h>
long send (netfd, destination, pbuff)
    int netfd;
    L_ADDRESS *destination;
    struct b_desc *pbuff;

Description
The primitive is used to transmit data to a named destination; at most 2 Gbytes of data can be transmitted. The parameter "netfd" is the file descriptor obtained by a successful open(2) applied to a special file associated with a specific network. The param-
eter “destination” provides the (logical) address of the destination process on that network. The pointer “pbuff” points at an array of buffer descriptors (see udsio(5)). Data are gathered from the buffers addressed by these descriptors and transmitted to the destination address as a single “datagram.”

This primitive returns a long integer indicating the number of bytes actually transmitted. It is to be regarded as an error if the return value of send(2) is not equal to the sum of the buffer lengths specified in the array of buffer descriptors pointed at by “pbuff.”

See Also
open(2), net_if(4), udsio(5)

Diagnostics
The return value −1 indicates that an error has occurred; the external variable “udserr” provides an indication of the type of error.

NET_IF(4)
Name
net_if - network interface description

Description
This section describes the network interface. This interface provides six primitives for (i) handling port numbers, (ii) transmitting and receiving datagrams, and (iii) disclosing the mapping between logical and real addresses. These primitives are all implemented as UNIX “ioctl” calls.

Each network to which a UNIX system is directly connected has associated at least one special file in the directory “/dev” of that system, in the same way as any other UNIX I/O device. An application can access a network by applying the standard UNIX open(2) operation to one of the special files associated with that network. The effect of this operation will be (i) to identify the driver of that network, and (ii) to return a file descriptor to be used by that application in any further operation on that network.

Processes using the K interface can either select their own port number, or obtain a dynamically allocated port number from the network driver. In addition, user processes may wish to select the source addresses from which they intend to receive data, and to fetch the “signature” of the received data (if any). To this end, the interface associates an instance of the data structure “net_if” (defined below) with each special file corresponding to a network. This data structure can be used by the user processes to pass information to the drivers, and to fetch information from the drivers.

The “net_if” data structure is defined as:

```c
struct net_if {
    unsigned long input_port; /*Input port number. */
    short qualifier; /*Accept selector. */
    L_ADDRESS source; /*Selected source address. */
    short signature[4]; /*Data signature. */
    short timeout; /*Receive timeout. */
    short neterror; /*Error indication on failure. */
    long netlength; /*Length last received message. */
    short host_maxl; /*Max. length real host address. */
    short port_maxl; /*Max. length real host sub-address. */
};
```

The “input_port” field can be set by a user process either to a value selected by that user process itself within a predefined range (see below), or to the value DYNAMIC (defined in <udsio.h>). In the former case, the driver will associate the selected port number with that process, if that port number is not in use by another process, otherwise the driver will return an error indication. If “input_port” is DYNAMIC, the driver will dynamically generate a port number for that process. (This port number can be fetched by invoking the primitive netget(2).) Input from the network directed to the port number will be delivered to the process associated with it.

DYNAMIC port numbers are always generated in the range [1025, ..., 65535]. Port numbers within the range [0, ..., 1023] are assigned by the Driver upon explicit request from the application software; thus, these port numbers can be used by the application software as “well-known addresses,” as they can be statically assigned to specific servers. The value 1024 is used as the representation of DYNAMIC and is never assigned.

The two fields “qualifier” and “source” of the “net_if” data structure allow a process to select the source addresses of the data that the process wishes to accept. The field “qualifier” can be set to one of the following five values (defined in <udsio.h>):

qualifier = ANY_SELECT indicates that the process wishes to accept data coming from any source address;

qualifier = HOST_PORT_SELECT indicates that the process wishes to receive only data com-
The field "netlength" contains the length of the last received message. The user may want to fetch this value, by invoking netget(2), when, for example, a receive(2) operation terminates unsuccessfully because of lack of buffer space in user memory.

Finally, the fields "host_maxl" and "port_maxl" indicate the maximum length, in number of bytes, of the real host address and sub-address. The user may want to fetch these two fields, by using netget(2), in order to allocate adequate buffers to hold real addresses.

See Also
netget(2), send(2), receive(2), litor(2), udsio(5)

**UDSIO(5)**

**Name**

udsi0 - constants and data types defined on network interface

**Synopsis**

include &lt;udsi0.h&gt;

**Description**

This file contains the constants and data types defined by the proposed network interface.

The following are the values which may be assumed by the "qualifier" field in the structure "net_if."

```c
#define NOONE 0
#define HOST_PORT_SELECT 1
#define HOST_SELECT 3
#define PORT_SELECT 5
#define ANY_SELECT 7
```

The following is the value of the constant DYNAMIC which may be assigned to the field "input_port" of the "net_if" structure so as to obtain a dynamically allocated port number from the driver. This value is not included in the range of DYNAMIC ports which may be assigned.

define DYNAMIC 1024 /*Obtain a dynamic port number. */

The following is the value to be assigned to the "timeout" parameter of the receive(2) primitive in order not to associate any timeout with this primitive.

define NOTIMEOUT 0 /*No timeout on receive(2). */

Application processes can use the send(2) and receive(2) primitives to transmit and receive the contents of a set of buffers residing in noncontiguous areas of their own memory, as defined by an array of buffer descriptors. A buffer descriptors is defined as:

```c
struct b_desc { /*Buffer descriptor. */
    char *p_buff; /*Pointer to buffer start address. */
    long b_length; /*Buffer length (in bytes). */
};
```
The field "p_buff" is a pointer to a buffer whose length, in number of bytes, is given in the field "b_length." The array of buffer descriptors is to be terminated by a buffer descriptor containing a NULL pointer. In transmission, data are gathered from these buffers and transmitted as a single "datagram"; in reception, a "datagram" received from the network is scattered to the various buffers whose addresses and length are specified in the array of buffer descriptors.

The send(2) and receive(2) primitives operate on addresses of type L_ADDRESS:

```c
typedef struct l_address {
    unsigned long host_number;
    unsigned long port_number;
} L_ADDRESS;
```

The field "host_number" identifies the address of a host computer on a network; the field "port_number" identifies a subaddress within that host.

Different NS Interfaces are likely to maintain real addresses characterized by different formats. We propose that real addresses be uniformly reflected on the K interface by the data type R_ADDRESS, defined below:

```c
typedef struct r_address {
    char *host_address;
    short h_a_length;
    char *sub_address;
    short s_a_length;
} R_ADDRESS;
```

The field "host_address" is a pointer to a sequence of bytes which is used to represent the real address of a host computer on a network (e.g., the DTE address of a host on an X.25 network, the station number of a host on a Cambridge Ring); the field "h_a_length" indicates the length, in number of bytes, of the sequence of bytes pointed at by "host_address." The field "sub_address" is a pointer to a sequence of bytes that is used to represent the real address of a process within the host at "host_address" (e.g., the name associated with a service on a Transport Service network, the port number associated with a process on the Cambridge Ring); the field "s_a_length" indicates the length, in number of bytes, of the sequence of bytes pointed at by "sub_address." In general "host_address" and "sub_address" may contain null bytes; they should not be considered as null-terminated strings.

It will be the responsibility of the application software to interpret these fields correctly (e.g., the "host_address" field may contain a number of characters to indicate a network number). Note that the use of this data type to represent real addresses is only a convention agreed upon by Adaptors and application software, i.e., it does not need to bear any relation to the internal representation of real addresses as maintained by an Adaptor.

The following variable holds the diagnostic error number when a network error occurs:

```c
int udserr /*Network error diagnostic.*/
```

See Also
ltn(2), send(2), receive(2)

Acknowledgement

The preparation of this paper, and the development of the interface it describes, has been greatly aided by our various colleagues at Newcastle. Our particular thanks go to A. Linton for the Cambridge Ring implementation of the interface, R. Stroud for the Ethernet implementation, G. Dixon for the RS232 implementation, J. Black and L. Marshall for their constant advice during both the design and the implementation, and S. Shrivastava for his constructive criticisms to our proposal. K. Bennett (University of Keele), P. Collinson (University of Kent), C. Wadsworth (SRCRuthford and Appleton Laboratories), and I. Wand and K. Ruttle (University of York) have also provided us with very useful feedback.

References


Fabio Panzieri received the “Laurea” degree in computer science from the University of Pisa, Pisa, Italy, in 1978, and the Ph.D. degree from the University of Newcastle upon Tyne, Newcastle upon Tyne, England, in 1985. From 1979 to 1985 he was a Research Associate with the Computing Laboratory of the University of Newcastle upon Tyne. He is now with Soma-Sistemi Vono Macchina, Rome, Italy. His research interests are in the areas of distributed systems, primitives for distributed computing, and communication protocols for local and wide area networks.

Brian Randell received the degree in Mathematics from Imperial College, London, England, in 1957.

He then joined the English Electric Company where he led a team which implemented a number of compilers, including the Whetstone KDP9 Algol Compiler. From 1964 to 1969 he was with IBM, mainly at the IBM Research Center in the United States, working on operating systems, the design of ultrahigh-speed computers and system design methodology. He then became Professor of Computing Science at the University of Newcastle upon Tyne, England, where in 1971 he initiated a program which now encompasses several major research projects sponsored by the Science and Engineering Research Council and the Ministry of Defence of England.