The Architecture of UNIX United

JAMES P. BLACK, LINDSAY F. MARSHALL, AND BRIAN RANDELL

Invited Paper

UNIX United is an architecture for a distributed system based on UNIX. As it is compatible with UNIX at the system call level, any program written for a normal UNIX system can be transparently extended to exploit the richer environment of UNIX United. As it relies on having a UNIX system beneath it, the implementation of UNIX United, called the Newcastle Connection, provides an interesting example of the construction of a very powerful distributed system with only a modicum of effort. A description of the basic semantics of UNIX United is followed by that of the architecture implied by the protocol between components in a UNIX United system, and of a software structure appropriate to the architecture and the protocol.

I. INTRODUCTION

UNIX United and the Newcastle Connection were first described in [5], which contained a quite extensive survey of work on UNIX-based distributed systems, and comparisons of the different approaches that have been adopted. No attempt is made to repeat such a survey in the present paper. Since that time, the two notions of UNIX United as an architecture and the Newcastle Connection as an implementation have become more distinct in our own minds, and both have evolved considerably in response to our continuing design and implementation efforts. The purpose of this paper is twofold: to describe the semantics and architecture of UNIX United in some detail, and to discuss the current state of our design and implementation.

A UNIX United system is composed of a number of component UNIX systems connected by one or more communications media. In architectural terms, UNIX United is a loosely coupled collection of components for a number of reasons: it should be feasible to use both fast and slow communications media, administrators of a component should retain their autonomy in the distributed system, and any given UNIX United system should be capable of encompassing an arbitrary number of components. While UNIX United is intentionally loosely coupled in the senses described above, it paradoxically presents an extremely integrated view to its users; that of a single, albeit very large, UNIX system in which all of the normal UNIX system calls and programs exhibit exactly the same behavior when executed in the UNIX United environment as when executed in the environment of a single, isolated component.

The result is that UNIX United is recursively structured [13]: the functionality of the distributed system as a whole is identical to that of its components. This not only has some interesting consequences in terms of the design of distributed computing systems, but it also implies that all existing software investments in UNIX can be retained in UNIX United, without necessarily requiring any modification to their source code or that of the UNIX kernels on the component machines. (As distributed commercially, the Newcastle Connection consists essentially of a replacement for the C language system call library, and thus programs only need to be relinked to be used in the UNIX United environment. However, we and others have also created UNIX United systems by installing the Newcastle Connection software below the physical machine kernel boundary, just "on top of" the essentially unmodified kernel. In this case, no change whatever is required to existing programs.) Clearly, this also implies that the user's perception of UNIX United is identical to his perception of UNIX itself; the advantages of this cannot be overstated.

In Section II, we discuss the motivation and basic semantics of UNIX United in more detail. Section III discusses the architecture of UNIX United, or precisely how the semantics of UNIX are extended in UNIX United. Section IV describes the software structures associated with the architecture, both in terms of our implementation (the Newcastle Connection), and in terms of the remote system call protocol which is used between various processes on UNIX machines in a UNIX United system. Section V discusses the current status of our project, and Section VI presents some conclusions.

II. MOTIVATION AND BASIC UNIX UNITED SEMANTICS

Our original reason for developing UNIX United was that we had available a number of small UNIX machines linked by a Cambridge Ring, but no particularly useful or appealing means of using them to provide us with a distributed system. Further, as we had no large pool of readily available manpower, we were forced to make as much use as possible of the UNIX software which we had, and were unable–or at least unwilling–to contemplate developing any significant amount of new software. These constraints, together with some of our own ideas on system design and aesthetics, provided much of the motivation and background for our initial development of UNIX United.

Thus one of the most distinctive aspects of our work is that we have constructed a distributed system whose program and user interfaces are indistinguishable from those of UNIX, and we have avoided re-implementing or modifying any existing UNIX facilities or mechanisms. We have provided network-transparent use not just of files, but also of all the other types of objects supported by UNIX, including processes, devices, pipes, and signals. This network transparency was added at the lowest level of UNIX visible to users and their programs: the system call interface. In performing our extension of UNIX to UNIX United, we made the working assumption that the UNIX system call interface is adequate for use in a distributed system, even though it was designed for a centralized system. (This is not to say that we in fact feel it is perfect for either environment.)

There are several reasons supporting our choice of the system call interface as a point where we could easily extend UNIX into a useful distributed system. First, system calls provide the ability to create and use multiple processes, and are also the sole means by which any process can interact with the world outside it, which consists of files, devices, other processes, etc. The fact that one can identify all interactions between each process and its environment makes it feasible to provide means for intercepting such interactions. These interactions, which in normal circumstances would all be handled within a single computing system, can then be transformed where necessary into interactions which involve communicating with other computer systems. Thus programs designed to use files on just one system can find themselves using files that are attached to some other system, and programs designed to use multiple processes can find themselves using multiple processors.

A second and less common characteristic that the UNIX system call interface happens to possess is of equal importance. This is that the major name space that it provides, i.e., its hierarchical file and directory naming scheme, is fully contextual, with directories acting as contexts. Any file or directory is named, directly or indirectly, relative to one or the other of two starting directories, namely, the root directory and the current working directory. However, neither of these can be located absolutely, since both are positioned beforehand by means of a system call that uses a context-dependent name. Moreover, files and directories outside a context can be named relative to that context using the convention that ".." indicates the parent directory. This avoids the need to know the name by which the context is known in its surrounding context. One can, therefore, readily construct an overall UNIX-like name space for the distributed system as a whole simply by combining together the name spaces of the constituent UNIX systems. This will be without any requirement to change existing names, and without fear of introducing any name clashes. (The root directories of individual component UNIX systems will be identified with directories in this overall UNIX United name space.)

In its full generality, then, a UNIX United name space is indistinguishable from a normal UNIX naming tree, although the former would tend to be larger than the latter. If this is truly the case, then there are two equally appropriate ways to view the construction of a UNIX United system. One way takes the point of view that a number of existing UNIX systems must be joined in some fashion which results in a global UNIX United naming tree. In this case, the system administrators of the components must come to some agreement about the structure of the intercomponent part of the naming tree. The other way takes the point of view that some large naming structure is a given initial condition (perhaps due to exterior organizational constraints), and the problem is to find some appropriate assignment of real UNIX component systems to this naming structure.

In either case, one of our important original goals was to permit the arbitrary placement of components in the UNIX United tree: in general, the path from one system to another may involve traversing any number of intermediate systems. However, this traversal is in naming terms only: it may well be the case that all the systems involved can in fact communicate directly over some single underlying communications medium.

Unfortunately, the UNIX United contextual naming scheme encompasses only some of the entities that UNIX and its users have to identify. UNIX also supports several simple ("flat") name spaces. A case in point is the scheme for identifying users themselves (either at the kernel level, by means of "user ids," or at the level of "login," for example, by means of "user names"). This scheme implicitly assumes that there is a single unstructured set of users, together with a centralized administrative procedure for authorizing new users, and ensuring the distinctness of their ids and their user names. In our view, this assumption is inappropriate for a distributed system of any great size. We have therefore made provision for UNIX United systems to be constructed from component UNIX systems, each with its own (potentially different) set of users, and even its own system administrator.

Essentially the same problem arises with other name spaces that UNIX uses internally, whether on a per-system basis, such as "process ids," or on a per-process basis, such as "file descriptors." For example, it would surely be inappropriate to demand that all the component UNIX systems somehow shared a centralized mechanism for generating new process ids, merely in order to avoid clashes. Thus the semantics of UNIX United must integrate the semantics of the global name space with other semantics of UNIX which are not directly tied to path names. Understanding these UNIX United semantics requires a reasonably precise definition of what constitutes one of the component systems in UNIX United. We thus define a component system to be a root directory, plus a set of user ids, a set of group ids, a set of process ids, and a local notion of time. Normally, a user would also assume that each system stored a number of standard programs, files (e.g., a password file), and devices (e.g., a console terminal), but these are required neither by the UNIX system call interface, nor by UNIX United. As indicated above, the naming tree of each component system becomes a subtree of the UNIX United naming tree, while explicit mappings must be maintained by the UNIX United implementation among the times and identifiers local to each machine.

Most of the semantics of files and directories are extended in the obvious manner: any path name used as an argument in a system call can be a pathname which leads to a remote system. However, because of the local nature of process and user identifiers, care must be taken to ensure that a running process only perceives identifiers appropriate to its root system, that is, the component system associated with the root directory of the process. As discussed more fully below, the importance of this is most obvious during remote execution of a program file. In such cases, actual execution may take place on a remote system, but any information about its environment which a process obtains through system calls is guaranteed to refer to the root system, which mayor may not be the system where execution occurs. This is necessary to ensure the internal consistency of all the data observed by a process.

Conceptually, then, a UNIX United system consists of what is potentially a very large number of component UNIX systems, whose naming trees have been joined together to form one very large naming tree. In order to guide our design and implementation of this distributed naming facility, we adopted an "infinite system" philosophy: each component should consider itself to be part of a UNIX United system which is so large that no single component can reasonably have global knowledge of all the other components participating in it. Put another way, our design problem was to find an appealing scheme for storing an "infinite" naming tree on a number of "finite" components. Our solution is based on the two ideas of a name neighbor and a name neighbor space. Given a UNIX United naming tree, two components in it are name neighbors if pathnames from one to the other do not pass through a third system. In general, there may be some number of directories in the path between two name neighbors, and these directories may often contain entries for other systems. A set of name neighbors forms a name neighbor space if every pair chosen from the set is a pair of name neighbors. Fig. 1 shows part of a universitywide UNIX United system. If the anonymous node at the top of the tree is a directory, and those just below it are systems, then {CS, EE, Math,. . .} is a name neighbor space. If the third-level nodes in the tree are also systems, then the leftmost U1 and U2 are name neighbors of CS but not of each other; in this example, CS is also a member oftwo other trivial name neighbor spaces, {CS, U1}, and {CS, U2}.

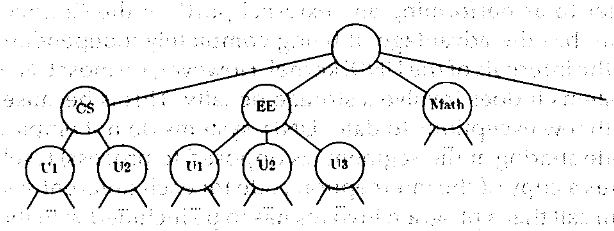


Fig. 1. A UNIX United system.

In UNIX United, each component stores its local naming tree, plus that part of the global tree between it and its name neighbors in all directions. Thus each member of a name neighbor space stores a copy of the directory structure between the name neighbors. This inter-name neighbor directory structure is created and maintained manually by the administrators of the components belonging to the name neighbor space, and is assumed to be as static as other important parts of the naming tree (such as standard program directories, the password file, etc.). During pathname interpretation, the local prefix of a path name (if any) is interpreted and stripped before the remainder is forwarded for processing at a name neighbor. As a consequence, every component operates independently of any other as long as all path names issued on it are local. In the example, CS, EE, and Math each store a copy of the directory above them. On the other hand, the U systems only store the upward link to their parent.

The size of a name neighbor space, and the character strings associated with its members, are a matter for negotiation between the administrators of the components involved. If, for example, one component belongs to a large number of two-component name neighbor spaces, then it is acting as a name server: any pathnames connecting components in different spaces pass through the name server, and only the name server component need record the names and communication paths linking the others. On the other hand, such an approach implies that the name server must be available for intercomponent communication to occur, and so it might be necessary to implement special fault-tolerance measurements within the hardware and software of the name server.

At the other end of the spectrum lie large name neighbor spaces. Here, the members act more as peers, as each stores a copy of all the directory information linking them together. This may have the advantage that any pair of members may communicate between themselves as long as they are both operating, but also has the disadvantage of perhaps requiring special mechanisms for ensuring the consistency of the name neighbor space and managing changes to it. Different regions of a single UNIX United system can reflect different choices, and the choice between replication and centralization can easily be changed by the assignment of a UNIX processor to, or its removal from, a particular node of an unchanging global naming tree. Thus the UNIX United architecture is neutral to this important tradeoff between replication and centralization in distributed system design.

As a UNIX United naming tree is identical to a UNIX naming tree, all of the protection and authentication facilities of normal UNIX are present in UNIX United. Additionally, when a user issues a path name that leads from one component to a name neighbor, the administrator of the name neighbor is given the ability to control all remote access to his system. Because user and group identifiers in UNIX are small integers, there is already a need to perform some sort of mapping when a user on one component performs operations on another. There may be no mapping for a particular incoming user, in which case he is refused access, or he may be given a default local user id, or he may be given a unique id, all at the discretion of the administrator. In general, we consider the authentication algorithm to be the responsibility of the system administrator, and so provide only a simple algorithm in our software. In principle, this could be carried to the exteme of providing different types of restricted server processes to different remote users, depending on information stored in the local mapping tables. Thus as the administrator can control the exact facilities made available to the remote user by UNIX United, he has in some sense more control over the actions of remote users than he might normally have over those of local users.

Note that while our discussions are phrased in terms of each component having its own administrator, it will often be the case that several components are in fact administered by a single person, and will in fact have a common user community. Architecturally, UNIX United makes no provision for grouping systems administratively. Practically, administering several systems with UNIX United is very much easier than trying to administer a number of isolated systems without the facilities of normal UNIX that UNIX United makes available across a set of systems. Again, UNIX United is neutral: it permits components and their administrators to be mutually suspicious, while at the same time providing convenient facilities (extended from UNIX to UNIX United) for administration of several systems by one administrative authority.

It is important to emphasise that the UNIX United architecture is defined only in terms of the UNIX system call interface, or more precisely, only in terms of the protocol used to encode UNIX system calls for transmission to and interpretation on a remote component. Thus it permits several kinds of diversity in the components making up a UNIX United system. First, the components can vary widely in their storage capacity, number of users, peripheral devices, and processor speed, conceivably ranging from lap portables to mainframes. Second, the protocol provides for explicit identification of variant system calls, which allows the interconnection of components running many different "UNIX" operating systems. Third, any computer which appears to respond reasonably to the protocol can participate in UNIX United: all that is required is the implementation of an interpreter for the protocol. An implementation may well return standard error codes for system calls that it is not prepared to emulate. The difficulty of writing such an interpreter depends on factors such as the similarity of the host operating system to UNIX and the number of system calls to be emulated.

In the section that follows, we describe the UNIX United architecture in more detail. Although the description assumes the availablilty of standard UNIX systems to support UNIX United, the reader should keep in mind that the only true requirement is that all communicants respond properly to the underlying protocol for encoding UN IX system calls.

III. Architecture

As indicated earlier, UNIX United systems are constructed by adding the Newcastle Connection software to each of a set of interlinked UNIX systems. The task of this software is to filter out system calls that have to be re-directed to another UNIX system, and to accept system calls that have been directed to it from other systems. Communication between systems is based on the use of a remote procedure call protocol. The general scheme is thus as portrayed, somewhat simplistically, in Fig. 2.

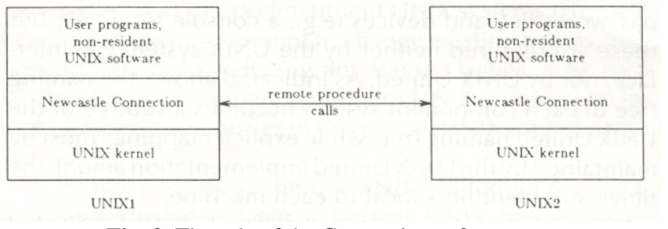


Fig. 2. The role of the Connection software.

In fact, the Connection software has three principal components. The interception layer performs the actual task of diverting appropriate system calls so that they are handled by a distant rather than the local UNIX kernel. Diverted calls are received by what we term a UNIX server. Such servers are created as needed by the so-called spawner. The resulting structure within each component system is, therefore, somewhat more accurately portrayed in Fig. 3.

In the rest of this section we describe these three major components, and how they are used to create what we term distributed sequential processes, using the remote procedure call protocol. For simplicity, we deal with issues relevant just to Version 7, briefly mentioning some of the complications introduced by subsequent versions of UNIX later.

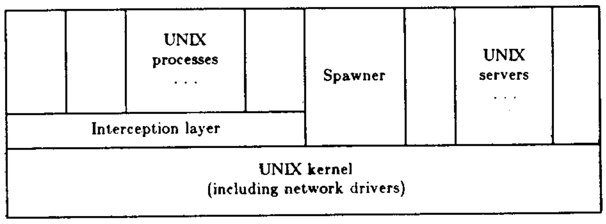


Fig. 3. Structure of the Connection software.

The interception layer was originally developed, and is made available to others, in the form of a system call library. To use it directly in this form, the code of each UNIX program that is intended to use UNIX United (i.e. that is to be "connected") has to be re-compiled, or at least re-linked. By this means any system call will actually invoke a routine from the interception layer which has the same name and visible functionality as the corresponding routine from the standard system call library. (The interception routine itself contains a call to the standard routine, to which it passes the parameters of any call that is to be handled locally.)

This method of installing the interception layer (which we refer to as performing an "external port" of the Connection) has the advantage of being completely independent of the internals of the UNIX kernel. However, on most UNIX systems it does involve a storage penalty. This is because, with few exceptions to date, UNIX systems do not employ code sharing at the segment (as opposed to process) level. Thus a copy of the interception code for each different system call that a program invokes has to be included with the program code. (The extra space involved therefore depends on the program, but is typically about 10 kbytes.)

This storage penalty, and the need to re-compile or re-link existing programs, can be avoided by installing the interception layer within the UNIX kernel. The main drawback of this technique is that it is kernel-specific. However such "kernel ports" of the Connection have now been successfully completed for several different UNIX systems. Nevertheless, our recommendation is that a kernel port should be treated as an optimization exercise, after undertaking an external port. This enables any problems resulting from nonstandard or additional system calls to be sorted out before tackling problems arising from the idiosyncrasies of the internal design of the kernel.

The way in which an interception routine processes a particular system call depends upon a number of factors such as the pathnames(s) passed to it, if any, the semantics of the system call itself, and the current internal state of the process as maintained by the Connection. In the simplest case, a path name leading to a remote system is passed as an argument. Some initial substring of the path name can be interpreted locally to yield a communications path to a name neighbor, and the remainder of the pathname to be interpreted there. Recursively, a UNIX server at the name neighbor will process the path name sent to it, and may also find that it leads to a yet more remote component. When this path name interpretation terminates, a true UNIX system call is made by a UNIX server on some component, and the results of it are returned to the originating user process.

Thus interpretation of a single system call may involve communication between the user process and one or more UNIX servers. Each UNIX server process is created to deal with a single client, and normally exists until the client process terminates. As the server runs with a true local user id determined by the mapping mentioned earlier, all the necessary protection and authentication is implemented by the remote UNIX kernel. On the other hand, if a single server process were to try to serve multiple clients, it would effectively be necessary to reimplement a significant fraction of the UNIX kernel itself within the server.

Clearly, the interception layer within the client process must maintain information about the number and location of server processes that it has created. When a UNIX server is not yet available for a client at a name neighbor, the interception layer sends a message to a "spawner" process at a well-known address, requesting that a server be created. The message includes information about the requesting process, enabling the spawner to consult its mapping tables to determine if the access should proceed at all, and if so, what the local user id of the UNIX server it creates should be.

We define a distributed sequential process (DSP) to be one user process together with its current collection of UNIX servers on some number of component machines. In fact, the distributed sequential process is the extension of a normal UNIX process to the UNIX United environment. Thus while it is, in fact, a cooperating set of processes, it appears, for all intents and purposes, to be a single sequential process.

Most simple path name and file descriptor calls are no more complicated than this. If only two systems are involved, on message is sent in each direction for each system call that must be executed remotely. In the case of file descriptors, the interception layer maintains a mapping between those seen by the client program, and those held locally or remotely. A simple table lookup suffices to indicate where the call must be executed. In the case of pathnames, more work must often be done to determine which part of the path name is local. Typically, this involves the use of one or more "stat" system calls, assuming that the root and current directory are local. If they are not, state information allows the entire path name to be forwarded without the necessity for issuing extra system calls.

System calls like "fork" are more complicated to implement than the simple cases above. "Fork" duplicates the calling process, and so the interception layer for "fork" must duplicate the entire distributed sequential process. This involves requesting a "fork" operation at each UNIX server currently in use by the process, and then issuing a true local "fork" to complete the operation. Care must be taken in creating new communications paths to the new servers, and in recovering from any errors observed before the entire sequence of operations is complete.

Once complete, "fork" is often followed by an "exec" operation to overlay the client program with another program specified by a path name argument. Roughly, if the target pathname is local, a local "exec" is performed, with some care being taken to pass the state of the interception layer through to the interception code in the overlayed program. In the more complex case, the target path name is remote, and so a more significant rearrangement of the distributed sequential process must be undertaken. As UNIX United is intentionally loosely coupled, it cannot be assumed that the target load module can be executed on the local processor: execution of the client must be transferred to the remote site.

However, "exec" by itself changes neither the current working directory nor the current root directory, nor for all intents and purposes the fact that some parent process may eventually wait on the termination of this process. Therefore, a "stub" UNIX server must remain at the current site, replacing the user component of the DSP. As the root directory has not changed, neither has the root system (including the set of user ids, the set of process ids, and the local notion of time), and so the remotely executing user code must perceive any such information from the environment of that root system. The execution environment is maintained by the interception layer. Finally, all the other UNIX servers belonging to the DSP must be informed of the rearrangement of the DSP: when a true "exec" is finally executed on the remote system, it will result in the user component of the DSP moving to the target system. Fig. 4 shows a DSP before and after an "exec" operation; the circle represents the user component of the DSP, initially on the root system, while the triangles represent UNIX servers. Note how the communications paths between the components of DSP (indicated by lines) are rearranged as well.

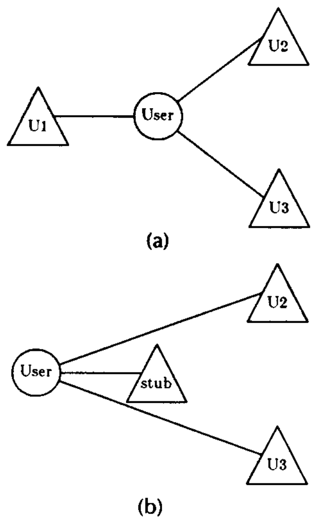


Fig. 4. (a) DSP before "exec." (b) DSP after "exec."

These semantics occasionally have unexpected consequences. For example, the "who" program prints a list of users currently logged in to the root system. Logically, then, "who" should return exactly the same output, regardless of which copy of the program might be executed on one of a number of components. This is because all of the user ids "who" observes should be those of the root system. More subtly, "who" actually produces its output by reading a file called "/etc/utmp," which is specified by a root-relative pathname. As the root has not changed, any remotely executed copy of the "who" program will open the same (local) file, and produce identical results. Only by such careful attention to the semantics of UNIX United can we ensure that it is truly transparent.

On the other hand, it might well be useful to know who is logged in to another component. In normal UNIX, there is a privileged "change root" system call which just might achieve such an effect. However, it has its own semantics in UNIX United (left as an exercise to the reader), and cannot be subverted for use in this case. Our solution is to add one new (pseudo-) system call, "excr" (execute with changed root). "Excr" behaves just like "exec," except that it also causes the root system of the DSP (including root directory, ids, and local time) to move to the last UNIX component encountered in tracing the target pathname. By executing "who" with "excr" instead of "exec," the result will be a list of people logged in to the target UNIX component, rather than to the root system of the issuing DSP. "Excr" is similarly useful in obtaining lists of processes active on other components, the true owner information for files (normally, such information on remote files is mapped back to user ids of the root component), and in general, for executing any program that issues root-relative path names which it tacitly assumes remain within the component where it is stored.

One of the aspects of UNIX which poses the greatest problems in UNIX United is that of signals issued between processes. At first glance, it seems clear that a "kill" system call to issue a signal must be executed on the root system (which may not be the system where the program is being executed and is stored). For signals causing process termination, however, the interception layer in the receiving process needs to trap the signal in order to ensure orderly shutdown of all UNIX servers belonging to the DSP. Similarly, a stub server receiving a signal destined for the user component of its DSP must trap the signal in order to forward it to the user component (via the spawner on that system). However, problems arise because not all signals can be trapped by a process, and hence, some signal semantics in UNIX United cannot be precisely correct without placing the interception layer mechanisms inside the UNIX kernel. (Associated with these objective problems are some more subjective feelings that the semantics and uses of signals are much less attractive and easy to understand than is the case for many other concepts in UNIX.)

IV. SUPPORTING SOFTWARE AND PROTOCOL STRUCTURE

The previous section presented those aspects of UNIX United relevant to processing individual system calls, which we have called the "server" level of the protocol. An equally important part of our architecture has to do with lower level facilities for transmitting information across a single network, maintaining inter-name-neighbor communication paths across multiple networks, and housekeeping functions associated with the distributed sequential process. In this section, we will discuss the "chain level" of the protocol, which provides an inter-name-neighbor remote system call facility to the level above, based on some underlying data transmission facilities.

Much of this work stems directly from the infinite system philosophy, which in this instance holds that a UNIX United system may be so large that no single communications medium can be used to connect all components. Among other things, this implies that no broadcast facilities can be used (or needed) by UNIX United, nor can we use an algorithm which enumerates all nodes or routes between nodes. Thus we have attempted to face the prospect of internetworking in a way which provides necessary functionality in the presence of multiple networks, and yet which avoids the necessity of implementing a "real" internetwork-a task we thankfully leave to those more competent and more wellendowed with time and money.

The lower part of the chain level (and hence the interception layer overall) assumes the availability of some means for transmitting single messages across a network; the transmission is not assumed to be completely reliable. We have called this abstraction of a network an "address space," and the idealized communication services provided across an address space are called the "Uniform Datagram Service" (UDS) [12]. Basically, an address space consists of a number of hosts, each of which supports a number of communication ports. Futhermore, every host in the address space appears to communicate equally easily with every other host in it. Hosts and ports are identified with fixed-length integers whose interpretation is only meaningful within the address space; this is in contrast to schemes such as that reported in [11], which assign much larger globally unique addresses to hosts.

Basic data transmission primitives are available for very long messages (on the order of megabytes). This allows the interception layer to treat long read and write requests without itself performing fragmentation and reassembly. Indeed, many existing network interfaces incorporate fragmentation and reassembly, as well as sophisticated protocols for error recovery using selective retransmission. When this is the case, the retransmissions that the chain level itself may cause will occur with negligible frequency. When the UDS address space itself appears unreliable, the cost of retransmission depends both on the frequency of errors and on the size of the message retransmitted, but our feeling is that our allocation of responsibility between the chain layer and the address space is cost-effective for a wide range of transmission speeds, network reliabilities, and message sizes. Our aim is to permit cost-effective implementations of UNIX United on a wide range of computing systems, such as small personal workstations connected by serial lines for which a minimal UDS interface can be developed in less than a man-month, or a number of large mainframes linked by wide-area networks and associated complex protocols. We have implemented a number of UDS address spaces, and feel that they are much simpler to use and understand than, say, sockets [10].

This UDS interface is then used as a basis for constructing a simple and more reliable remote procedure call protocol which is intended only for sending UNIX system calls between the parts of a distributed sequential process. This protocol has evolved from that described in [16]. Most significantly, it has been simplified and specialized to be useful only between the components of a single DSP. Thus no global clock synchronization is required, as all RPCs are issued sequentially by the user component. A simple integer sequence numbering scheme suffices to detect duplicates and permit retries to be attempted if no reply is received. Some attempt is made to ensure that reuse of port numbers by the network driver cannot result in mistaken acceptance of messages destined to or originating from the wrong process.

One of our early design choices was that the UNIX United naming tree should be independent of the underlying physical communications topology. Two components which happen to be name neighbors may well be several address spaces apart, while components on the same address space may well be hierarchically organized into many name neighbor spaces. On the other hand, it is important to be able to take advantage of physical proximity whenever possible. For all system calls involving pathnames, UNIX semantics require that permissions be checked at each directory (and hence each component system): communication is necessary with all component systems along the path name, in spite of possible physical proximity of the final system encountered in tracing it. For system calls such as "read" and "write" which do not have pathname arguments, however, efficiency demands that only the initial and final systems on the original pathname be involved, whenever this is permitted by the available communications media.

For example, suppose that the UNIX United system of Fig. 1 is supported by two address spaces: CS, EE, and systems below them in the naming tree share a local area network, while EE also belongs to a network linking Math and other systems not shown below it and to its right. Consider a process with its root on U3, which issues an "open" system call with a pathname "/../../CS/U1/file." UNIX semantics require that permissions be checked at U3, EE, CS, and U1; UNIX United semantics imply that the spawners at these last three systems may also permit or deny access to the issuing process. However, once the "open" has succeeded, "read" and "write" system calls may proceed directly from U3 to U1 without the intervention of the UNIX servers at EE and CS. On the other hand, any system call issued by the same process to a system at or beyond Math would need to be forwarded through a server on EE, as no single address space connects all the systems in the diagram.

In the first instance, then, the chain level must support communication between name neighbors across one or more UDS address spaces. In the general case, the communication information stored locally for each name neighbor includes which address space to use for the first hop, and an uninterpretable token to present to the spawner at the end of the first hop. The spawner recursively uses this token to determine the outgoing address space and new token. This results in the creation of a chain of relay servers at each physical neighbor along the path to the UNIX server at the name neighbor. For example, if the process on U3 possessed several UNIX servers in parts of the naming tree below Math, communications with them would all be handled by the same server on EE, which would be acting both as a UNIX server for pathnames through EE, and as a relay server for nonpathname system calls to more remote systems. (We point out, however, that the mechanism described also makes it possible for a system to provide relay service even though it is not mentioned in a path name issued by a process.)

Another important responsibility of the chain level is general housekeeping for the DSP. For example, it implements file descriptor optimization so that the file descriptor returned by an "open" operation on a remote system can later be used for 1/0 without the need to involve multiple UNIX servers on the same address space. Briefly, this is done by associating a physical address with each return message, and then ensuring that this physical address is appropriately modified when the message passes through a relay server. Other responsibilities of the chain level include reorganization of the DSP in response to "exec" system calls, process termination, process creation, and new server creation; all these facilities are chain operations rather than "new" system calls visible at the server level.

As the chain level is only concerned with inter-nameneighbor communication, it is occasionally unable to perform some optimization which might be possible with a global view of the DSP. For example, if a path name through several name neighbors and address spaces eventually leads back to an earlier address space, this will not be recognized. Such phenomena are consequences of our conscious decision to attempt only a simple form of internetworking. Another way of stating the problem is to say that if the naming tree and the physical communications topology are too poorly matched, then this architecture will not be optimal. We feel that it is necessary that our architecture functions correctly under such circumstances, but not necessary that it functions optimally. In cases where bandwidth requirements or organization of the address spaces presented to the chain level result in unacceptable performance, the clear alternative is to implement a proper internetwork layer, encapsulate it within a higher level UDS address space, and do all the required optimizations below, rather than within, the chain level.

Generally, each UNIX system call results in just one message being transmitted in each direction, as the result of a system call is used to acknowledge its reception. There are exceptions to this due on the one hand to some UNIX semantics, and on the other hand to the presence of multiple networks.

For example, system calls such as read and write may take an arbitrarily long time to complete if the device involved is a terminal under human control. Our solution in this case involves the sending process making some small number of retries, and if this fails, checking that the target UNIX server is still alive. If so, the process then issues an arbitrary number of retries with timeouts larger by an order of magnitude, checking for the existence of the server at the expiry of each. This "infinite retry" logic is controlled by a switch passed as an argument from the server level.

In cases where a UNIX system call must be sent through an intermediate server before arriving at the target system, designing remote procedure call timeouts for each hop is problematical. Note that the intermediate server may be a UNIX server if the call is crossing more than one name neighbor space, or a relay server if it crosses more than one address space. To resolve this problem, individual calls may optionally be acknowledged at the chain level by the receiver, in which case the sender also enters the infinite retry logic described above.

V. Present status

Pre-release versions of our software were first made available to other organizations in mid-1982. The first formal release was issued in June 1983 and since then a considerable number of organizations have taken out either commercial or educational licenses, and have ported the Connection to various other machines, networks, and versions of UNIX. The current release of the Newcastle Connection provides support for Unix System III, System V, and BSD 4.2, as well as Version 7. It has thus been necessary to include a regrettably large amount of conditionally compiled code to deal with the differences between these UNIX "variants," and to provide at least some degree of interworking between them. Currently the principal limitations are that only a single name neighbor space, and a single (possibly virtual) network address space are supported, although prototype versions of the Connection have been constructed that relax both these restrictions.

Adding support for System V was relatively simple, involving only the addition of some new system calls and only slight modifications to others-mainly in order to encompass incompatible changes made in the name of standardization! However, some of the new system calls do present the problem that their interface is not suitable for generalization to a distributed environment. A case in point is the "ustat" system call (introduced with System III) which takes as one of its parameters a system-dependent value that "names" one of the devices attached to the machine. This value can be obtained from the information returned by the "stat" system call and so a user could use a value that was only meaningful on a remote system. It is possible to overcome this on a per-process basis, but a general solution (without changing the interface) would seem to be impractical or even impossible. Moreover, no attempt has been made to implement the shared memory operations, which in any case are described in [2] as being hardware-dependent, and not necessarily present on all systems.

In contrast, BSD 4.2 involved considerably more work, not only because of the large number of new system calls that had to be supported, but also because of the desire to take advantage of some of the new facilities provided by these calls. (For example, the use of the "select" system call allowed the removal of a large amount of code concerned with simulating nested timeouts.) However, the new signal interface was the biggest cause of changes, in the form of additional compiled code (without significant benefit to the Connection!). The only part of BDS 4.2 not fully supported at the moment is the socket interface, calls on this being forbidden in remotely executing programs. This is due to the difficulty of building RPCs for calls such as "send," where the structure of the data passed as parameters cannot be determined simply, due to the variety of network interfaces that may be supported via sockets.

Another recent complication concerns the use of pathnames starting with "/..". In UNIX United remote systems are just directories which can appear anywhere in the directory hierarchy. If system B is represented by a directory below "/" on system A, then on system B, system A must be referred to as being above B's "/" if circularity is to be prevented and a proper hierarchy guaranteed. Unfortunately, the latest versions of UNIX enforce a rule that "/.." is always the same as "/," and this effectively prevents such directories, which have been our preferred means of constructing an overall directory hierarchy. On such versions of UNIX it is necessary to create special i-nodes representing remote systems below "/," and consequently it is possible to construct circular path names. This should only be a problem with commands like "find /" which traverse the entire directory tree. In fact, since the current release of the Connection only supports a single name neighbor space, UNIX servers do not forward system calls (are not "transitive"), so one cannot make references from one remote system to another remote system. Thus infinite loops are prevented at the cost of a violation of the transparency provided by the Connection. Plans are in hand to allow transitive UNIX servers, at which time we intend to incorporate mechanisms into the Connection which will effectively prevent such apparent circularities causing problems.

One of the earliest ports of the Connection was to the PERQ running a derivative of UNIX System III called PNX. This gave us our first experience of dealing with the problems arising from the use of bit-mapped graphics and window managers. On PNX, windows are treated as devices, so read and write calls between windows on separate machines are handled by existing Connection mechanisms. The main graphics primitive, "RasterOp," is a new system call, and caused few complications, but the amount of information that may have to be transmitted is, of course, a severe test of the underlying network facilities. The principle problems arose from calls concerned with window control, yet are not really specific to windows. It is not easy to add a new type of object, such as a window, to UNIX–traditionally, this has been done using device drivers and "ioctl" calls. Unfortunately, the parameter to "ioctl" is essentially typeless, and so it is difficult to determine quite what information constitutes the parameter so that it can be transmitted with an RPC to a remote machine. (BSD 4.2 has added some limited typing facilities which go someway towards solving this problem.)

Turning finally to issues regarding the performance of a UNIX United system, it is clear that this depends on three essentially separate factors: the capabilities of the component UNIX systems, the efficiency of the underlying communications hardware and software, and the overheads due to the Connection, only the last of which is our responsibility. In fact, the overheads due to the Connection are really quite modest. The effect on local system calls involving just file descriptors is imperceptible, though local pathname calls such as "open" and "exec" are slowed down somewhat, since for each such call an additional "stat" system call is made from within the Connection code. When a call involves a remote object, this normally just involves making one RPC call, and waiting for a reply. In the absence of network errors, RPCs only involve two messages. The UNIX server that accepts such calls from remote machines in most cases does little more than make a single call for its client and send it the results.

As reported in earlier papers, our first UNIX United system, which was based on PDP11s and a Cambridge Ring, functioned surprisingly well, despite the fact that the Ring stations used were quite slow, being interrupt-driven rather than direct memory access devices. (Such stations cause UNIX to take an interrupt for every pair of bytes sent and received over the Ring!) In fact, terminal users in general noticed little performance difference between local and remote execution and file accesses. This perhaps indicates that even interrupt-driven Ring stations are reasonably well matched to the rather modest performance that UNIX itself can achieve on a small PDP11123 used as a personal workstation, or on a PDP11/45 being used by a number of demanding terminal jobs.

We also performed some simple measurements using our PERQ-based system, which has much more adequate network hardware, in fact an Ethernet with direct memory access interface units. These measurements produced the initially surprising result that copying files to or from a remote PERQ could be at least 20 percent faster than local file copying. In fact, this merely indicates the extenttowhich contention for a single disk can limit a workstation's performance. One other interesting measurement showed that file transfers using the standard UNIX file copy command and the Connection achieved almost twice the speed achieved by the manufacturer-supplied file transfer protocol, which uses the ECMA Level 4 Transport Service [1] over the Ethernet. However, the more significant result is that, on this system also, users in general notice little difference in performance between local and remote operations.

The most recent evidence we have of the performance of the Connection has been obtained from a small series of experiments on VAX/750 computers, connected by Ethernet. These experiments compared the use of the Connection with the use of the remote copy and remote login facilities that are provided with BSD 4.2. Fig. 5 shows the time taken to copy files of various sizes from the local machine to a remote one, either using the Connection and the standard "cp" command, or the Berkeley "rcp" command. It will be seen that (connected) "cp" achieves over twice the speed of "rcp" on small files, and that only with files whose length exceeds 100 kbytes does "rcp" start to show even modest performance advantages. (We acknowledge that elapsed time is not an ideal measure; however, neither are the locally observed values for system and user time, as they exclude time spent waiting for remote operations. The figures shown are averages of 100 trials on a normal 4.2BSD system, with no other processes active except the usual daemons.)

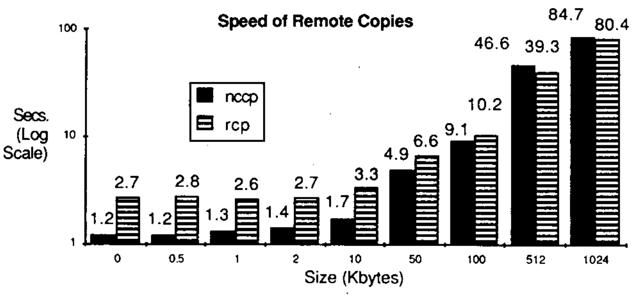


Fig. 5. File transfer comparison.

A separate project has now been set up at Newcastle to undertake detailed performance monitoring of UNIX United systems at the system call level. The aim of this project is both to study the way in which a UNIX United system is used, and to obtain a more detailed assessment of the performance of the Connection.

Turning to questions of storage occupancy, the total amount of code involved in the various parts of the Connection is about 11000 lines of C. Of this, the UNIX server code amounts to approximately 2500 lines, the code involved in intercepting and mapping the various system calls some 7500 lines, and the "spawner" which is used to start up the file server processes on demand the remaining 1000 lines. Installation of the Connection as a separate layer involves including a copy of selected parts of the interception code in each user program. On the VAX under 4.2BSD, for example, the size of the load module increased by between 5.7K ("echo") and 15.3K (the Bourne shell), depending on the number of different system calls that the program invokes. For a sample of 18 common programs including the two mentioned above, the average increase was 9.6K. (The alternative means of installing the Connection, discussed briefly in Section III, involves inserting just the interception code in the kernel-the file server and spawner code remain outside the kernel.)

VI. Concluding Remarks

UNIX United and the Newcastle Connection result, we would claim, from a successful blend of opportunism and, dare one say, philosophy–the former stemming from our sudden realization that UNIX provided both an appropriate semantics for a general-purpose distributed system, and appropriate mechanisms and interfaces for this system to be constructed merely by adding a comparatively simple transparent subsystem to UNIX. The design philosophy we employed was at the outset little more than an active concern for structure and generality, and more particularly, a liking for recursive constructs (dating back to work at Newcastle on recursive virtual machines [9], if not earlier). However, as a result of our work on the Connection, these ideas on recursive system structuring have become much more well-defined, in our own minds at least, and have enabled us to separate carefully issues concerned with constructing a distributed system from those concerned with taking advantage of the fact that it is distributed, for example in order to provide increased reliability, availability, and/or security. This is not to say that we have simply ignored all such issues. Rather we have investigated, and in several cases already implemented, various separate but complementary reliability and security mechanisms, each of which can simply be added to a UNIX United system, without requiring modifications to the code of either UNIX or the Connection [3], [4], [14]. (This work is surveyed in [13], as part of a general account of our ideas on recursive structuring.)

Although the Connection is designed to support loosely coupled systems, as we have explained, it also allows the construction of systems for use in environments in which a higher degree of coupling is required, for example where there is a single community of users and where centralized control is exercised by a single system administrator. In either case, the system can be used exactly as if it were a conventional UNIX system–an illusion which can be particularly realistic when the component systems are linked by a LAN, since the bandwidth of the LAN will typically exceed that of their disk storage access channels. The advantages of this illusion, to both users and system administrators alike, have proved to be very considerable.

The other illusion that we have started to explore involves the use of the UNIX system call interface as a means of linking heterogeneous components together. In effect, this involves using the RPC protocol as a virtual operating system, linking together a mixture of UNIX and non-UNIX systems into a coherent whole. Quite how far such an approach can be taken successfully remains to be seen–to date, the most extensive attempt to exploit this idea has been the design and implementation of a terminal concentrator, based on a message-passing kernel, which appears to the rest of UNIX United as a UNIX system which merely contains a set of "/dev/tty" names. However, others at Newcastle have explored the use of Connection-style linking of various types of microcomputers, and the aim is now to permit the incorporation of such systems directly into a UNIX United system.

It would be inappropriate to end these concluding remarks without an explicit acknowledgment of our debt to UNIX and its original creators–it has its deficiencies, of course, both as a centralized system, and as the basis of a general-purpose distributed system. Nevertheless, we have found its facilities, particularly at the system call level, and the style of system design that it exemplifies, a veritable inspiration. Such simplicity and generality of mechanism as we have been able to achieve undoubtedly owes much to this source.

ACKNOWLEDGMENT

D. Brownbridge and L. F. Marshall were involved in the very early design of UNIX United. L. F. Marshall has borne the major responsibility for its design and implementation, aided recently by J. P. Black. The Remote Procedure Call Protocol on which that now used is based was originally designed and implemented by F. Panzieri and S. Shrivastava. The UDS interface was designed by F. Panzieri and B. Randell, and has been implemented for various types of machine and network by F. Panzieri, A. Linton, R. Stroud, and G. Dixon. R. Stroud and D. Sharp have been responsible for the first kernel ports of the Connection. We are grateful for the collaboration and encouragement of several people at the University of Illinois at Urbana-Champaign, who were instrumental in developing a version of the Connection for BSD 4.2 [8], and later implementing a kernel port [15]. They have also made some use of the Connection in support of teaching [7], and of a research project [6].

REFERENCES

[1] European Computer Manufacturer's Association, *Standard ECMA-72 Transport Protocol,* Jan. 1981.

[2] AT&T, "UNIX System V-Release 2.0 Programmer Reference Manual in DEC Processors," IT&T publ. 307-113, Issue 2, Apr. 1984.

[3] J. A. Anyanwu, "A reliable stable storage system for UNIX," *Software–Practice and Experience*, vol. 15, no. 10, pp. 973-990, Oct. 1985.

[4] J. A. Anyanwu and L. F. Marshall, "A crash resistant UNIX file system," *SoftwarePractice and Experience*, vol. 16, no. 2, pp. 107-118, Feb. 1986.

[5] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection-of UNIXes of the world unite!" *Software–Practice and Experience*, vol. 12, no. 12, pp. 1147-1162, Dee. 1982.

[6] R. H. Campbell and W. J. Kubitz, "The professional workstation project," *IEEE Compo Graphics Appl.*, vol. 6, no. 5, pp. 17-24, May 1986.

[7] M. V. Devarakonda, R. E. McRath, R. H. Campbell, and W. J. Kubitz, "Networking a large number of workstations using UNIX United," in *Proe. 1st Int. Conf. on Computer Workstations*, pp. 321-339, Nov. 1985.

[8] J. M. Donnely, "Porting the Newcastle Connection to 4,2 Berkeley UNIX," M. Sc. Thesis, Dept. Computer Sci., University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-1199, Urbana, Il, Mar. 1985.

[9] H. C. Lauer and D. Wyeth, "A recursive virtual machine architecture," in *Proc. ACM Workshop on Virtual Computer Systems*, pp. 113-116, Mar. 1976. Also available as University of Newcastle upon Tyne, Computing laboratory, Tech. Rep. TR54.

[10] S. J. Leffler, R. S. Fabry, and W. N. Joy, "A 4.2BSD interprocess communication primer," Dept. Elec. Eng., Computer Systems Research Group, Univ. of California, Berkeley,July 1983.

[11] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. ACM*, vol. 19, no. 7, pp. 20-66, July 1976.

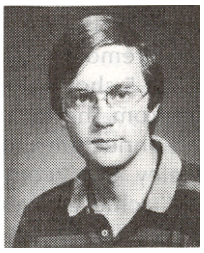
[12] F. Panzieri and B. Randell, "Interfacing UNIX to data communications networks," *IEEE Trans. Software Eng.*, vol. SE-11, no. 10, pp. 1016-1032, Oct. 1985.

[13] B. Randell, "Recursively structured distributed computer systems," in *Proc. Symp. on Reliability in Distributed Software and Database Systems*, pp. 3-11, Oct. 1983.

[14] J. M. Rushby and B. Randell, "A distributed secure system," *IEEE Computer*, vol. 16, no. 7, pp. 55-67, July 1983. Also available as University of Newcastle upon Tyne, Computing laboratory, Tech. Rep. TR182.

[15] V. F. Russo, "A kernel implementation of UNIX United," M. Sc. thesis, Dept. Computer Sci., Univ. of Illinois at Urbana Champaign, Urbana, Il, 1985.

[16] S. K. Shrivastava and F. Panzieri, "The design of a reliable remote procedure call mechanism," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 692-697, July 1982.



James P. Black received the Ph.D. degree in computer science from the University of Waterloo, Waterloo, ant., Canada, in 1982, having previously studied at the University of Calgary, Calgary, Alta., Canada, and the Institut National Poly technique de Grenoble, Grenoble, France. He spent two years as an NSERC/NATO Postdoctoral Fellow at the Computing laboratory of the University of Newcastle upon Tyne, Newcastle upon Tyne, U.K., where he participated in the development of a distributed system called the Newcastle Connection. He is currently an assistant professor at the University of Waterloo, where he is pursuing his research interests in robust data structures, software reliability, and distributed systems.

Lindsay F. Marshall received the B.Sc. (Honors) degree in 1974 from the University of Edinburgh, Edinburgh, U.K., and the Ph.D. degree in 1980 from the University of Newcastle upon Tyne, Newcastle upon Tyne, U.K., both in computer science. From 1977 to 1981 he was employed by the British Ship Research Association, where he worked on CAD/CAM systems for use in the shipbuilding industry. Presently, he is a research associate with the Computing laboratory of the University of Newcastle upon Tyne. His research interests include the design of highly reliable systems and their user interfaces.

Brian Randell graduated from the University of London, London, U.K., in 1957. He joined the English Electric Company, where he led a team that developed a number of compilers, including the Whetstone KDF9 Algol Compiler. From 1964 to 1969 he was with IBM, mainly at the T. J. Watson Research Center, Yorktown Heights, NY, working on high-performance computers, multiprocessing systems and system design methodology. In 1969 he returned to the U.K. as Professor of Computing Science at the University of Newcastle upon Tyne, where he initiated a program of research on system reliability which now encompasses a number of related research projects, sponsored by the U.K.

Science and Engineering Research Council and the Royal Signals and Radar Establishment.