THE NEWCASTLE CONNECTION:

a software subsystem for constructing distributed UNIX* systems

B. Randell

Computing Laboratory,
University of Newcastle upon Tyne

ABSTRACT

The Newcastle connection is a software subsystem that can be added to each of a set of physically interconnected UNIX or UNIX look-alike systems, so as to construct a distributed system which is functionally indistinguishable at both the user and the program level from a conventional single-processor UNIX system. The techniques used are applicable to a variety and multiplicity of both local and wide area networks, and enable all issues of inter-processor communication, network protocols, etc., to be hidden. A brief account is given of experience with such distributed systems, the first of which was constructed in 1982 using a set of PDP11s running UNIX Version 7, and connected by a Cambridge Ring - since this date the Connection has been used to construct distributed systems based on various other computers and versions of UNIX, both at Newcastle and elsewhere. The final sections compare our scheme to various precursor schemes and discuss its potential relevance to other operating systems.

1. INTRODUCTION

The Newcastle Connection is the name we have given to a software subsystem which enables a distributed system to be constructed out of a set of standard UNIX systems. Such distributed systems (which can use a variety and multiplicity of both local and wide area networks) are functionally indistinguishable, at both `shell' command language level and at system call level, from a conventional centralised UNIX system [1]. Thus all issues concerning network protocols, and inter-processor communication are completely hidden. Instead all the standard UNIX conventions, e.g. for protecting, naming and accessing files and devices, for inter-process communications, for input/output redirection, etc., are made applicable, without apparent change, to the distributed system as a whole.

The Newcastle Connection can be installed without any modification to any existing source code, of either the UNIX operating system, or any user programs. The technique is therefore not specific to any particular implementation of UNIX, but instead is applicable to any UNIX look-alike system that claims, and achieves, compatibility with the original at the system call level.

In subsequent sections we discuss the structure of such distributed systems, (which for the purposes of this paper we will term UNIX United systems), the internal design of the Newcastle Connection, the networking issues involved, some interesting extensions to the basic scheme, our operational experience with it to date, its relationship to prior work and its potential relevance to other operating systems.

2. UNIX UNITED

A UNIX United system is composed out of a (possibly large) set of inter-linked standard UNIX systems, each with its own storage and peripheral devices, accredited set of users, system administrator, etc. The naming structures (for files, devices, commands and directories) of each component UNIX system are joined together in UNIX United into a single naming structure, in which each UNIX system is to all intents and purposes just a

*UNIX is a Trademark of Bell Laboratories.

directory. Ignoring for the moment questions of accreditation and access control, the result is that each user, on each UNIX system, can read or write any file, use any device, execute any command, or inspect any directory, regardless of which system it belongs to. The simplest possible case of such a structure, incorporating just two UNIX systems, is shown below.

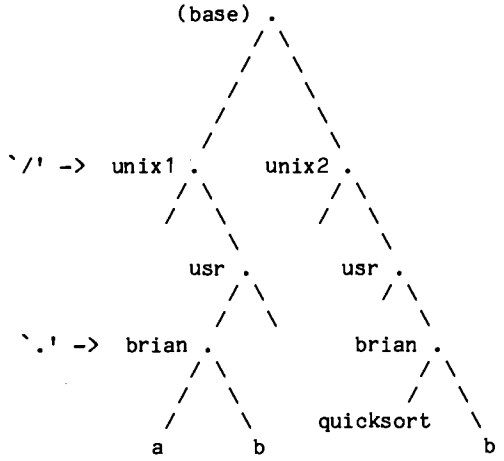

Figure 1: A Simple UNIX United System

With the root directory (`/') positioned as shown, one could copy the file `a' into the corresponding directory on the other machine with the shell command

```
cp    /user/brian/a  /../unix2/user/brian/a
```

(For those unfamiliar with UNIX, the initial `/' symbol indicates that a path name starts at the root directory, and the `..' symbol is used to indicate the parent directory.)

Making use of the current working directory (`.') as shown, this command could be abbreviated to

```
cp   a  /../unix2/user/brian/a
```

If the user has set up the shell variable `U2' as follows

```
U2=/../unix2/user/brian
```

it could be called forth, using the `$' convention, so as to permit the further abbreviation

```
cp   a $U2/a
```

All the above commands are in fact conventional uses of the standard `shell' command interpreter, and would have exactly the same effect if the naming structure shown had been set up on a single machine, with `unix1' and `unix2' actually being conventional directories.

All the various standard UNIX facilities (whether invoked via shell commands, or by system calls within user programs) concerned with the naming structure carry over unchanged in form and meaning to UNIX United, causing inter-machine communication to take place as necessary. It is therefore possible, for example, for a user to specify a directory on a remote machine as being his current working directory, to request execution of a program held in a file on a remote machine, to redirect input and/or output, to use files and peripheral devices on a remote machine, etc. Thus, using the same naming structure as before, the further commands

```
cd /../unix2/user/brian

quicksort a > /../unix1/user/brian/b
```

have the effect of applying the quicksort program on unix2 to the file `a' which had been copied across to it, and of sending the resulting sorted file back to file `b' on unix1. (The command line

```
/../unix2/user/brian/quicksort /../unix2/user/brian/a > b
```

would have had the same effect, without changing the current working directory.)

It is worth reiterating that these facilities are completely standard UNIX facilities, and so can be used without conscious concern for the fact that several machines are involved, or any knowledge of what data flows when or between which machines, and of which processor actually executes any particular programs. (Programs are actaually executed by the processor in whose file store the program is held, and data is transferred between machines in response to normal UNIX read and write commands.) Moreover all standard UNIX facilities, even the system call used to reposition the root directory, are provided in UNIX United.

In fact what we have done in UNIX United is take advantage of an important but unusual property that UNIX possesses: all file naming is context-relative, in the sense that one can only name files relative to either the current or the root directory, both of which can be re-positioned (but again only using context-relative names). There is thus no way of naming files relative to any absolute point, such as the base of the tree. This feature of UNIX is more commonly used to enable multiple UNIX file systems to be held within one machine, but it is equally useful for splitting up a file system over a number of machines.

2.1. User Accreditation and Access Control

UNIX United allows each constituent UNIX system to have its own named set of users, user groups and user password file, its own system administrator (super-user), etc. Each constituent system has the responsibility for authenticating (by user identifier and password) any user who attempts to log in to that system.

It is possible to unite UNIX systems in which the same user identifier has already been allocated (possibly to different people). Therefore when a request, say for file access, is made from system `A', of system `B', on behalf of user `u', the request arrives at `B' as being from, in effect user `A/u' - a user identifier which would not be confused with a local user identifier `u'. It will be, in effect, this user identifier `A/u' which governs the uses by `u' of files, commands, etc., on machine `B'.

Just as the system administrator for each machine has responsibility for allocating ordinary user identifiers, so he also has responsibility for maintaining a table of recognised remote user identifiers, such as `A/u'. If the system administrator so wishes, rather than refuse all access, he can allow default authentication for unrecognised remote users, who might for example be given `guest' status - i.e. treated as if they had logged in as `guest', presumably a user with very limited access privileges.

From an individual user's point of view therefore, though he might have needed to negotiate not just with one but with several system administrators for usage rights beforehand, access to the whole UNIX United system is via a single conventional log in. Subject to the rights given to him by the various system administrators, he will then be governed by, and able to make normal use of, the standard UNIX file protection control mechanisms in his accessing of the entire distributed file system. In particular there is no need for him to log in, or provide passwords, to any of the remote systems that his commands or programs happen to use. This approach therefore preserves the appearance of a totally unified system, without abrogating the rights and responsibilities of individual system administrators.

At the other extreme, so to speak, it is possible to use the mechanisms we have provided to set up a UNIX United system in which there is, in effect, just a single system administrator, and a single set of accredited users. Then any user can sign on, in the same way, to any of the UNIX systems, and the system administrator can readily control, and perform system maintenance tasks relating to, the entire UNIX United system.

2.2. The Structure Tree

The naming structure of the UNIX United system represents the way in which the component UNIX system are inter-related, as regards naming issues. When a large number of systems are united, it will often be convenient to set up the overall naming tree so as to reflect relevant aspects of the environment in which the UNIX systems exist. For example, a UNIX United system set up within a university might have a naming structure which matches the departmental structure.

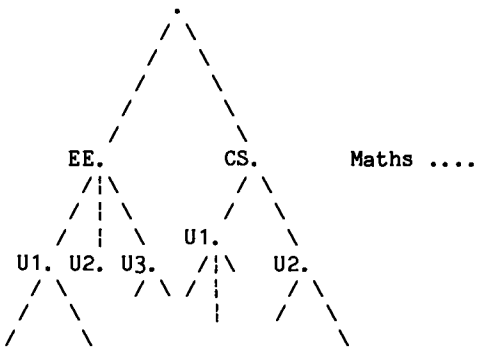

Figure 2: A University-wide System

With the naming structure as shown, files in the system `U1' in the Computing Science Department could be named using the prefix `/../../CS/U1' from within the Electrical Engineering Department's UNIX systems.

Such a naming structure has to be one that can be agreed to by all the system administrators, and which does not require frequent major modification - such modification of the UNIX United naming structure can be as disruptive as a major modification of the structure inside a single UNIX system would be, due to the fact that stored path names (e.g. incorporated in files and programs) could be invalidated.

The naming structure could, but does not necessarily, reflect the topology of the underlying communications network. It certainly is not intended to be changed in response to temporary breaks in communication paths, or of service from particular UNIX systems. (An analogy is to the international telephone directory - the UK country code (44) continues to exist whether or not the transatlantic telephone service is operational.)

One final point: We have developed mechanisms which make it possible for UNIX systems to appear in the naming structure in positions subservient to other UNIX systems, though these are not yet incorporated in the version of the Connection which we make available to other organizations. For example, in the previous figure, CS might denote a UNIX system, not just an ordinary directory. We regard this as a very important generalisation, since it allows existing UNIX United systems to be combined together, just as if they were ordinary UNIX systems.

3. THE NEWCASTLE CONNECTION

The UNIX United scheme whose external characteristics were described above is provided by means of communication links, and the incorporation of an additional layer of software - the Newcastle Connection - in each of the component UNIX systems. Conceptually, this layer of software sits on top of the resident UNIX kernel, i.e. between the UNIX kernel and the rest of the UNIX software (e.g. shell and the various command programs) and the user programs. In actual fact, one has a choice between keeping the Connection completely separate from the kernel, or of installing it within the kernel. The former is the simpler means of installing the Connection, but involves the recompilation or relinking of existing user programs and non-resident UNIX software. The latter technique is a kernel-specific optimization that avoids the need for such recompiliation or relinking. This method of installation naturally requires more effort and experience, and is best undertaken after completing the simpler porting technique, but in practice has not proved overly difficult. For convenience, in what follows we will assume that the Connection has been installed as a software layer, separate from the kernel.

From above, the Connection layer is functionally indistinguishable from the kernel. From below, it appears to be a normal user process. Its role 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 the Connection layers on the various systems is based on the use of a remote procedure call protocol [2], and is shown schematically below:

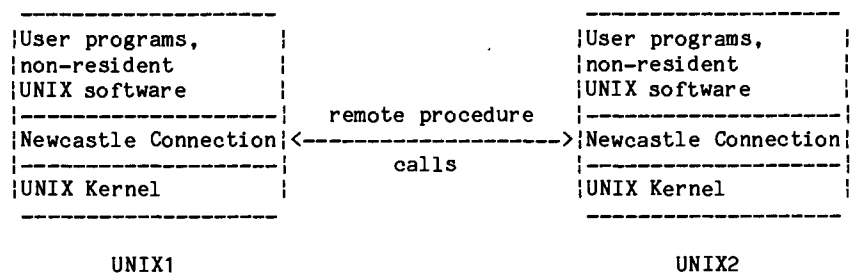

Figure 3: The Position of the Connection Layer

In fact a slightly more detailed picture of the structure of the system would of course reveal that communications actually occur at hardware level, and that the kernel includes means for handling low level communications protocols.

The Connection layer has to disguise from the processes above it the fact that some of their system calls are handled remotely (e.g. those concerned with accessing remote files). It similarly has to disguise from the kernel below it that the requests for the kernel's services, and the responses it provides, can be coming from and going to, remote processes. This has to be done without in any way changing the means by which system calls (apparently direct to the UNIX kernel) identify any real or abstract objects that are involved.

The kernel in fact uses various different means of identification for the various different types of object. For example, open files (and devices) are identified by an integer (usually in the range 0 to 19), logged on users by what is effectively an index into the password file, etc. Such name spaces are of course inherently local. The Connection layer therefore has to accept such an apparently local name and use mapping tables to determine whether the object really is local, or instead belongs to some other system (where it may well be known by some quite different local name). The various mapping tables will have been set up previously - for example when a file is opened - and for non-local objects will indicate how to communicate with the machine on which the object is located. The selection of actual communication paths is performed by the Connection layer, and completely hidden from the user and his programs.

Such mapping does not however apply to the single most visible name space used by UNIX, i.e. the naming structure used at shell level, and at the program level in the `open' and `exec' system calls, for identifying files and commands, respectively. Rather, the Connection layer can be viewed as performing the role of glueing together the parts of this naming structure that are stored on different UNIX machines, to form what appears to be a single structure. Each component UNIX system stores, firstly, the section of the naming tree associated with the system's own files and devices. Secondly, each system also stores a copy of those parts of the overall naming structure that relate it to its "name neighbours". These are the other UNIX systems with which it is directly connected in naming terms (i.e. which can be reached via a traversal of the naming tree without passing through a node representing another UNIX system).

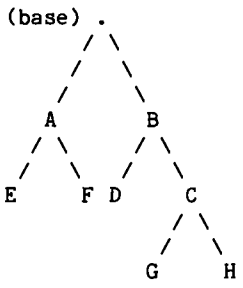

Figure 4(a): A UNIX United Name Space

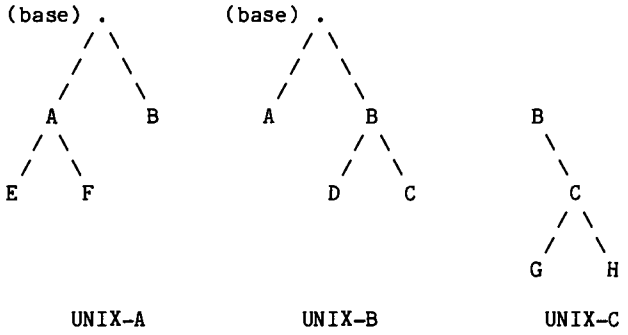

Figure 4(b): Represention of the Name Space

In Figure 4(a), if "directories" A, B and C are associated with separate UNIX systems, the parts of the tree representation stored in each system are as shown in Figure 4(b), namely:

UNIX-A: A,B,E,F,(base)

UNIX-B: A,B,C,D,(base)

UNIX-C: B,C,G,H

It is assumed that shared parts of the naming tree are agreed to by the administrators of each of the systems involved, and do not require frequent modification - a major modification of the UNIX United naming structure can be as disruptive as a major modification of the naming structure inside a single UNIX system. This is because names stored in files or incorporated in programs (or even just known to users) may be invalidated. (Again one can draw a useful analogy to the telephone system. Changes to international and area codes would be highly disruptive, and are avoided as far as possible. For example, they are not changed merely because the underlying physical network has to be modified.)

Within each UNIX system, the Connection uses the local fragment of the UNIX United naming tree to resolve file names. Names are interpreted as a route through the tree, each element specifying the next branch to be taken. If the name can be fully interpreted locally, only a local access is involved. If a leaf corresponding to a remote system is reached, then execution must be continued remotely by making a remote procedure call to the appropriate system. Such leaves are specially marked, and contain the network address of the appropriate remote station. This address is given to the RPC as routing information. (In some cases a request may be passed on through a number of Connections before being satisfied.)

As well as accessing files using a name, a UNIX program can `open' a file and thereafter access it using the file descriptor returned from the `open' system call. When a file is opened the Connection makes an entry in a per-process table indicating whether or not the file descriptor refers to a local or a remote file. The table also holds network station addresses for remote file descriptors. Subsequent accesses using the

descriptor refer to this table using the information there to route remote accesses without further delay.

The actual remote file access is carried out for the user by a file server process that runs in the remote machine. Each user has their own file server, and the initial allocation of these is carried out by a "spawner" process that runs continuously. This latter process is callable (using a standard name) by any external user, and, upon request, will spawn a file server (after carrying out some user/group mapping), returning its external name to the user that initiated the request. The user then communicates directly with this file server, which is capable of carrying out the full range of Unix system calls. The user/group mapping is carried out to ensure that the access rights of the file server are in accord with those allowed to the external user by the local system manager, and consists of converting external names into valid local names. Nevertheless, a file server is still an extension of the environment of a user on a remote machine, and any relevant changes in the environment seen by a user must be mirrored by it. The most important of these is that when a user process "forks" (that is, creates a duplicate of itself), all the remote file servers which it is connected with must also fork. This greatly simplifies the implementation of remote execution and signalling, as each user process only ever has to deal with a single remote file server.

Communication with the "spawner" and the file servers always takes the form of a remote procedure call, the first parameter of all calls being a sequence number. This is used by the servers to detect retry attempts - if the received sequence number is the same as that of the last call, then it is a retry (the RPC scheme precludes calls being lost, so there is no need to check for continuity in the sequence).

4. NETWORKING ISSUES

As indicated above, all communication between machines in a UNIX United system is performed by means of the RPC protocol, using network addresses that have been obtained from the leaves of the local fragment of the naming tree. Ideally, all the machines will be directly connected together, i.e. will belong to the same network address space, so that any machine can make a remote procedure call directly on any other machine. This is immediately achieved by the use of a single physical network, such as an Ethernet, but could be also be achieved by some sort of inter-network transport service which hides the existence of multiple physical networks from the Connection. However the Connection is currently being extended to contain its own provisions for coping with multiple network address spaces, and for forwarding RPC calls across networks, for use in situations where such a transport service is not provided.

Any actual implementation of the RPC protocol requires primitive operations for exchanging messages between processes on different machines. In order to shield the Connection layer from the complexities of having to handle differing network interfaces (for reasons both of simplifying its design, and improving its portablity) we have recently defined a single interface, the UDS interface, which provides a uniform process-to-process datagram service [3]. This hides the actual protocols used over each local or wide area network, and provides instead a small set of simple primitives for sending and receiving (possibly large) datagrams, using a standardized network addressing scheme, based on a <host number, port number> pair. Exception reporting is also standardized, the assumption being that the implementation of the UDS interface contains, where necessary, sufficient fault tolerance measures for the Connection to be able to rely on a datagram being transmitted accurately, and in its entirety, unless an exception is reported, in which case the Connection can request a retry. (The fault tolerance measures taken by any particular implementation will depend on the assumptions that can be made about the inherent reliability of the actual network involved.)

The UDS interface enables a datagram to be sent from, or received into, a set of non-contiguous buffers, and in effect places no further limit on the size of the datagram than that implied by the total size of the buffers. Each I/O driver implementing the interface for a given network therefore has the responsibility for performing any fragmentation and reassembly operations made necessary by the limitations imposed by the underlying communications hardware and software. This shields the rest of the Connection from such limitations, which in any case tend to be network-specific. The scatter/gather facilities provided to enable the use of non-contiguous buffers greatly reduce the amount of data copying involved in an RPC, and the ability to send large datagrams similarly reduces the number of I/O calls that have to be made across the user/kernel interface. Thus we have found that the introduction of the UDS interface in place of more

conventional network-specific datagram interfaces has not only simplified the tasks of the Connection, and the problems of porting it onto new hardware, but also given useful performance benefits [4].

5. EXTENSIONS TO THE BASIC UNIX UNITED SCHEME

We have found that the conceptual simplifications to the task of implementing a UNIX-based distributed computing system that the Newcastle Connection approach has provided have spurred us to produce a variety of extensions of, or variations on, the basic theme, some of which we have already started to implement.

The Connection layer can be regarded as isolating and solving the problems associated just with distribution - and, it turns out, is applicable to the case of distributed systems made from components other than complete UNIX systems. For example, one could connect together some systems which have little or no file storage with other systems that have a great deal - i.e. construct a UNIX United system out of workstations and file servers. Almost all that is necessary is to set up the naming tree properly.

Moreover since the Connection layer can be independent of the internals of the UNIX kernel, it is not even necessary for the Connection layer to have a complete kernel underneath it - all that is needed is a kernel that can respond properly (even if only with exception messages) to the various sorts of system call that will penetrate down through, or are needed to support, the Connection layer. In fact the Connection layer itself can be economised on, if for example it is mounted on a workstation that serves as little more than a screen editor, say, and so has only a very limited variety of interactions with the rest of the UNIX United system. All that is necessary is adherence to the general format of the inter-machine system call protocol used by the Newcastle Connection, even if most types of call are responded to only by exception reports.

Thus the syntax and semantics of this protocol assume a considerable significance, since it can be used as the unifying factor in a very general yet extremely simple scheme for putting together sophisticated distributed systems out of a variety of size and type of component - an analogy we like to make is that the protocol operates like the scheme of standard-size dimples that allow a variety of shapes of LEGO children's building blocks to be connected together into a coherent whole.

In addition to the problem of distribution, we also have taken what are, we believe, several other equally separable problems, in particular those of (i) providing error recovery (for example in response to input errors or unmaskable hardware faults), (ii) using redundant hardware provided in the hope of masking hardware faults, (iii) the enforcement of multi-level security policies and (iv) load balancing between the component systems, and plan wherever practicable to embody their solutions in other separate layers of software. Indeed, three significant extensions of UNIX United have already been implemented, albeit in prototype form. The first of these provides multi-level security, using encryption to enforce security barriers beween component machines (which each run at a single security level) and to control permissible security re-classifications [5]. The other two extensions are related to hardware fault tolerance. One uses file and process triplication and majority voting to mask hardware faults - application programs are unchanged, though in fact running in synchronisation on several machines with hidden voting. The other uses duplicated disks to provides a crash-resistant high integrity file system [6].

6. OPERATIONAL EXPERIENCE

The first UNIX United system was based on a set of three PDP11/23s and two aged PDP11/45s, all running UNIX V7, and connected by a Cambridge Ring. At the time of writing (August 1984) this system is being upgraded, with the 11/45s being replaced by VAX/750 computers. The system has been operational for over two years, and usually in regular daily use. Our experience has been that the most heavily used facilities have been those concerned with file transfer and I/O redirection, for example in order to make use of the line printer and magnetic tape unit that are attached to one machine. The Connection has also been relied on for network mail, for solving the problems of overnight file-dumping (of all machines, onto the one tape unit) and, perhaps most significantly, for software maintenance and distribution within the UNIX United system

A second and hitherto separate UNIX United system at Newcastle, based on ICL PERQs connected by Ethernet, was implemented during May 1983 in collaboration with ICL, and has

since also been used regularly. Work is now in hand to link this system to the VAX/750 computers, and to several other recently acquired UNIX machines, from various manufacturers, in order to produce one single enlarged and somewhat heterogeneous UNIX United system, involving both a Ring and an Ethernet.

Pre-release versions of our software were first made available to several other organisations, starting in mid-1982, the first formal release being issued in June 1983. By now 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, including System III and Berkeley 4.2.

As regards 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. Those caused by the need to confirm that a system call only involves a local file descriptor are virtually imperceptible, though local path name 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. When a call proves to involve a remote facility, this normally just involves making one RPC call, and waiting for a reply. The RPC protocol is itself very simple and usually involves sending one message (the packaged system call) and receiving one message in reply. The file server that accepts such calls from remote machines is similarly simple, being dedicated to serving the needs of just a single remote process, and in most cases does little more than make system calls on behalf of this process and send it the results.

Our first UNIX United system functioned surprisingly well, despite the fact that the Cambridge 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 accesses and execution. This perhaps indicates that even interrupt-driven stations are reasonably well matched to the rather modest performance that UNIX itself can achieve on a small PDP11/23 used as a personal workstation, or on a PDP11/45 that is usually being used by a number of demanding terminal jobs.

A separate project has now been set up to undertake performance monitoring and evaluation of UNIX United systems, a task whose difficulty derives in part from the well-know problems of making meaningful performance assessments and comparisons of ordinary UNIX systems. However some simple experimental measurements have already been made 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 of files to or from a remote PERQ could be 20% or more faster than local file copying. In fact this merely indicates the extent to which contention for a single disk can limit a machine'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 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 total amount of code involved in the various parts of the Connection is about 11,000 lines of C. Of this, the file 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 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 PDP 11/45, for example, the amount of code added varies between 3.5k and 12k bytes, depending on the number of different system calls that the program invokes. (The alternative means of installing the Connection, discussed briefly in Section 3, involves inserting just the interception code in the kernel - the file server and spawner code remains outside the kernel.)

On the PDP 11/45 the file server code occupies about 12.5k bytes, and in addition each actual server process requires 2.5k bytes of space. The single spawner process requires a total of 8k bytes of code and data space. (By way of comparison, the UNIX kernel as set up for our particular I/O configuration occupies 48.5k bytes of code space, and 83.5k bytes in total.) The comparatively small size of the Connection reflects the need we

had to make the system work on our small PDP11/23s, which provided a strong incentive to find what we feel justified in claiming are simple well-structured solutions to the various implementation problems. (In our view an overabundance of program storage space can have almost as bad an effect on the quality of a software system as does inadequate space - it is surely no coincidence that UNIX was first designed for quite modestly sized machines!)

7. RELATED EARLIER WORK

The Newcastle Connection, and the UNIX United scheme that it makes possible, have many precursors, and not just within the UNIX world.

The idea of providing a layer of software which aims to shield users of a set of inter-connected computers from the need to concern themselves with networking protocols, or even the fact of there being several computers involved, is well-established. It is, for example, what the IBM CICS System [7] does for users of various transaction-processing programs, and what the National Software Works project [8] aimed to do for the users of various software development tools, running on a variety of different operating systems. Such layers of software are intended for somewhat specialised use, and run on top of specific sets of application programs. At the other end of the spectrum, such location- or network-transparency is also one of the aims of the Accent kernel [9], on which operating systems can be constructed which use its "port" concept as a means of unifying inter-process communication, inter-computer message passing, and operating system calls.

The dawning realisation that the `shell' job control language and the program-level facilities (i.e. system calls) of the UNIX multiprogramming system could suffice, and indeed would be highly appropriate, to control a distributed computing system can be traced in a whole series of distributed UNIX projects. The global file naming technique used in the early `uucp' facilities [10] for interconnecting UNIX systems via standard telephone circuits can be seen as a special, but rather ad hoc, extension of the individual file system naming hierarchies, and had been copied by us in our Distributed Recoverable File System [11]. (The technique provides what is in effect a set of named hierarchies, rather than a single enlarged hierarchy.)

Rather better integrated with the standard UNIX file naming hierarchy are the facilities provided in the Network UNIX System [12]. This modification of standard UNIX provides a series of Arpanet protocols, which are invoked by means of some additional system commands, using what appeqr to be ordinary file names as the means of identifying which Arpanet host is to be communicated with. (The paper describing this system also speculates on the possibility of redesigning the shell interpreter so as to provide network transparency for commands and files at the shell command language level.) The Purdue Engineering Computer Network [13] is conceptually similar to the Network UNIX System, though based on hard-wired high speed duplex connections. It provides additional commands which invoke the services of special protocols for virtual terminal access and remote execution at the shell level, and also a means of load balancing through a scheduling program which takes responsibility for deciding which processor should execute certain selected commands.

The distributed system of interconnected S-UNIX personal workstations and F-UNIX file servers [14] goes further by providing each workstation user with a ordinary UNIX interface, without any additional non-standard commands, yet incorporating a distributed version of the UNIX hierarchical file store containing just his own local files plus all the files held on the file servers. This system is one of several built at Bell Labs using the Datakit virtual circuit switch - others are RIDE [15] and D/UNIX [16]. The RIDE system provides complete remote file access and remote program execution, but is based on a `uucp'-like, rather than standard, UNIX naming hierarchy - it is however implemented merely by adding a software layer on top of the UNIX kernel, an approach which is highly similar to that we have since used with our Newcastle Connection technique. D/UNIX is a distributed system based on modified versions of UNIX which provide virtual circuits between processes, and a transparent file sharing scheme covering all the files on all the component systems.

A fully symmetrical means of linking computer systems together so as to give the appearance of a single UNIX-like hierarchical file store, and the standard shell command language, is also provided by the LOCUS system - the paper [17] describing this system also discusses its intended extension to provide remote program execution as well as remote file access. However, for all its external similarity to UNIX, the LOCUS system

involves a completely redesigned operating system rather than a modification of an existing UNIX system, albeit an operating system which is also designed to have extensive fault tolerance facilities.

The penultimate stage in the evolution can be seen in the COCANET local network operating system [18], a system which has been built using the standard UNIX system, and which comes very close indeed to our aim of combining a set of standard UNIX systems into a single unified system, and which certainly supports network-transparent remote execution as well as file access. However the COCANET designers have allowed themselves to make a number of changes to the UNIX kernel and would appear, from the description they give, not to have coped fully with user-id mapping. It would also appear that COCANET is designed specifically around the idea of having a relatively small number of machines linked by a single high-speed ring, and hence has a rather restrictive structure tree, which is viewed slightly differently from each machine. However many of the mechanisms incorporated in UNIX United are very similar to those used in COCANET.

It is thus but a comparatively small step from COCANET to UNIX United, and to the idea of the Connection layer capable of being placed on top of an unchanged UNIX kernel, replicating all its facilities exactly in a network-transparent fashion, and capable of making a distributed system involving large numbers of computers, connected by a variety of local and wide area networks. Incidentally, one can draw an interesting parallel between the Connection layer and what is sometimes called a "hypervisor", the best-known example of which is VM/370 [19]. Each is a self-contained layer of software, which makes no changes to the functional appearance of the system beneath it (the IBM/370 architecture in the case of VM/370, which fits under rather than on top of the operating system kernel). However whereas a hypervisor's function is to make a single system act as a set of separate systems, the Newcastle Connection (a "hypovisor"?) makes a set of separate (though of course linked) systems act like a single system!

However, to our embarrassment, we have to admit that the idea of the Connection layer, of the basic UNIX United scheme, and of most of the extensions of the scheme, did not arise from careful study and analysis of these precursors. (Indeed it is clear that what was presented above as a more-or-less orderly evolutionary development path often involved parallel activity by several groups, and much accidental re-invention.) In fact we were not consciously aware of any of these systems (other than `uucp' and of course DRFS) whilst the work that led to the Newcastle Connection was in progress. Indeed, by the time we learnt of LOCUS and COCANET, all the basic ideas and strategies to be incorporated in the Newcastle Connection had been worked out, though not all in full detail, and much of the system was already operational and in daily use. Rather we can trace the origins of our scheme to the existence of the plans for our remote procedure call protocol, and the idea, which we now know has occurred to many groups independently, of extending the UNIX `mount' facility from that of mounting replaceable disk packs to that of mounting one UNIX system on another.

This idea arose at Newcastle in early December 1981 - within a week or so much of the UNIX United concept had been thought up and even roughly documented. A hesitant start at what was initially intended as just an experimental and partial implementation was made after Christmas, but within a month many facilities related to accessing and operating on files remotely over the Cambridge Ring were in active use. Work proceeded rapidly, both on extending the range of UNIX kernel features that the Newcastle Connection mapped correctly, and on discovering, mainly via experimentation, some of the more arcane features of the kernel interface as implemented and used in V7 UNIX. At about this stage we found out about first the S-UNIX/F-UNIX and LOCUS systems, and shortly afterwards the COCANET and then the RIDE systems. These various papers were a considerable encouragement to us to continue our efforts, leading to a first complete system by mid-1982, and also provided us with a useful perspective on our approach. In particular they strengthened our growing belief in the viability of an alternative, UNIX-based, approach to distributed computing to that based on the use of a variety of explicit servers, each with its own specialised service protocol [20,21].

8. WHY JUST UNIX?

It is interesting to analyse just what it is about UNIX, and the linguistic interfaces it provides at shell and system call level, that make it so suitable for use as the model and basis for a network operating system. There seem to be six principal factors involved.

First, there is the hierarchical file (and device and command) naming system. This makes it easy to combine systems, because the various hierarchical name spaces just become component name spaces in a larger hierarchy, without any problems due to name clashes. The standard UNIX mechanisms for file protection and controlled sharing of files then carry over directly, once the problem of possible clashes of user identifiers is handled properly.

Second, there are the UNIX facilities for dynamically selecting the current working directory and root directory. In particular the ability to select the root directory - normally thought of as one of the more exotic and little needed of the UNIX system commands - seems to have been designed especially for UNIX United, since it provides a perfect way of hiding the extra levels of the directory tree that have to be introduced.

Third, and obviously vital, is the fact that UNIX allows its users, and their programs, to initiate asynchronous processes. This is used inside the Newcastle Connection, and also provides the means whereby even a single user can make use via the Newcastle Connection of several or indeed all of the computers that are involved in the UNIX United system. It also provides the means whereby slow file transfers (via low bandwidth wide area networks) can be relegated to background processing, and so still be organised using remote procedure calls.

Fourth, there is the fact that the UNIX system call interface is (relatively) clean and simple, and can easily be regarded as providing a small number of reasonably well defined abstract types. The task of virtualising these types, so as to give network transparency, therefore remains manageable.

Fifth, there is the fact, even in this day and age still regrettably worthy of mention, that the original UNIX system, and all of its derivatives known to us, are written in a fairly satisfactory high level language. Our method of incorporating the Newcastle Connection into UNIX as a separate software layer therefore merely involved recompiling relevant parts of the system, using a different subroutine library.

Finally, there is the well-established set of exception reporting conventions that are used in UNIX, for example, to indicate the reasons why particular system call requests cannot be honoured. When such a call has, via the Newcastle Connection, involved attempted communication with another UNIX system there are various other (quite likely) reasons, but they can be mapped onto the exceptions that the caller is already supposed to be able to deal with.

However it is unlikely that the idea could not be carried across to at least some other systems. Indeed a report by Goldstein et al [22] implies that something similar is being bravely contemplated for IBM's MVS operating system, and some aspects of the idea are we understand commercially available as additions to the RSX/11 operating system - no doubt other examples exist. The one other system whose suitability for the Newcastle Connection approach has been considered at all seriously by us is DEC's VAX/VMS system. It would appear that it has many of the necessary characteristics though there could be problems with the way that devices are involved with its system of file naming.

This section would not be complete without any mention of what we regard as some shortcomings of the UNIX V7 specifications (at system call level): Firstly, the system of `signals' for asynchronous communication between processes could be improved. Allied to this, a general synchronous inter-process communication mechanism would be useful, allowing communication between numbers of unrelated processes. Some awkward features in the file protection scheme were encountered when constructing the file server. These were associated with the notions of `super-user' and `effective user-id'. Lastly, we found that the ability to have many directory entries (`links'), each naming the same physical file, was elegant in concept but severely limited in generality by the actual UNIX V7 implementation. (Unfortunately, the various "improved" versions of UNIX that have appeared since Version 7 have done little to remedy most of these shortcomings.)

With respect to the programs that are provided with the UNIX system, very few difficulties were encountered in connecting them, except, that is, for the Shell. This program makes use of system facilities in non-standard ways, and its internal design is obscure to say the least. However, it has proved to be an excellent testbed for the system - when porting the Connection, once the Shell works you can be pretty sure that most other programs will!

9. CONCLUSIONS

The first of our internal memoranda on what we later came to call the Newcastle Connection described the idea as "so simple and obvious that it surely cannot be novel". And, as described above, it did turn out to have a number of precursors - in fact probably many more than we yet realise. However we take this as confirmation of the merits of the twin ideas of network transparency and of its provision by a single separate mapping layer, an approach whose ramifications we feel we have barely begun to explore. (A more general discussion of this approach, and its relevance to the problems of designing highly reliable and secure systems can be found in[23].) Certainly our present plan is to continue our programme of experimental implementations and applications, and to determine how well the Newcastle Connection (and UNIX) can withstand the weight of additional software layers containing the various reliability and security-related mechanisms that we have developed, hitherto in a rather fragmented fashion for various systems and languages.

One other point is worth stressing. It has for some years been well-accepted that the structure and mechanisms of a multiprocessing operating system are very similar to those of a (good) multiprogramming system. What has now become clear to us, as a result of our work on UNIX United, is that this similarity can usefully extend also to distributed systems. The additional problems and opportunities that face the designer of a homogeneous distributed system should not be allowed to obscure the continued relevance of much established practice regarding the design of multiprogramming systems.

10. ACKNOWLEDGEMENTS

These lecture notes are based closely on the original paper describing the Newcastle Connection [24]. My co-authors on this paper, Dave Brownbridge and Lindsay Marshall, jointly implemented the pre-release version of the Connection, since when Lindsay Marshall has borne the major responsibility for its design and implementation, aided recently by Jay Black. The Remote Procedure Call Protocol on which that now used in the Connection is based was originally designed and implemented by Fabio Panzieri and Santosh Shrivastava. The UDS interface was designed by Fabio Panzieri and myself, and has been implemented for various types of machine and network by Fabio Panzieri, Andy Linton, Robert Stroud and Graeme Dixon. Robert Stroud has also had the major responsibility for the first port of the Connection (to the PERQ) and for its integration into a UNIX kernel.

A special acknowledgement is obviously due to the creators of that famous Registered Footnote of Bell Laboratories, UNIX, much of whose external characteristics, if not detailed internal design, deserve the highest praise. Last but not least, I am pleased to acknowledge that our work has been supported by research contracts from the UK Science and Engineering Research Council, and the Royal Radar and Signals Establishment.

* * *

References

[1] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", Comm. ACM Vol. 17(7), pp.365-375 (1974).

[2] S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism", IEEE Trans. on Computers (July 1982).

[3] F. Panzieri and B. Randell, "Interfacing UNIX to Data Communication Networks", Report 190, Computing Laboratory, University of Newcastle upon Tyne (Dec. 1983).

[4] A. Linton and F. Panzieri, "A Communication System Supporting Large Datagrams on a Local Area Network", Report 191, Computing Laboratory, University of Newcastle upon Tyne (1984).

[5] J. M. Rushby and B. Randell, "A Distributed Secure System", Computer Vol. 16(7), IEEE (July 1983).

[6] J. A. Anyanwu, "A Reliable Stable Storage System for UNIX", Report 191, Computing Laboratory, University of Newcastle upon Tyne (1984).

[7] J. Gray, "IBM's Customer Information Control System (CICS)", Operating System Review Vol. 15(3), pp.11-12, ACM (July 1981).

[8] R. E. Millstein, "The National Software Works: A Distributed Processing System", Proc. ACM 1977 Annual Conference, Seattle, Washington, pp.44-52, ACM (Oct. 1977).

[9] R. Rashid, "Accent: A Communication Oriented Network Operating System Kernel", Operating Systems Review Vol. 15(5), pp.64-75 (Dec. 1981).

[10] D. A. Nowitz, "Uucp Implementation Description", p. Sect. 37 in UNIX Programmer`s Manual, Seventh Edition, Vol. 2 (Jan. 1979).

[11] M. Jegado, "Recoverability Aspects of a Distributed File System", Software Practice and Experience Vol. 13(1), pp.33-44 (Jan. 1983).

[12] G. L. Chesson, "The Network UNIX System", Operating Systems Review Vol. 9(5), pp.60-66 (1975). Also in Proc. 5th Symp. on Operating Systems Principles.

[13] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A UNIX-Based Local Computer Network with Load Balancing", Computer, pp.55-66 (Apr. 1982).

[14] G. W. R. Luderer, H. Che, J. P. Haggerty, P. A. Kirslis, and W. T. Marshall, "A Distributed Unix System Based on a Virtual Circuit Switch", Proc. 8th Symp. Operating System Principles, Pacific Grove, Calif., pp.160-168, ACM (Dec 1981). Also in: ACM Special Interest Group on Operating Systems - Operating Systems Review, Vol 15, No 5 (Dec 1981).

[15] P. M. Lu, "A System for Resource Sharing in a Distributed Environment - RIDE", Proc. IEEE Computer Society 3rd COMPSAC, IEEE New York (1979).

[16] J. C. Kaufeld and D. L. Russell, "Distributed UNIX System", in Workshop on Fundamental Issues in Distributed Computing, ACM SIGOPS and SIGPLAN (15-17 Dec. 1980).

[17] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System", Operating Systems Review Vol. 15(5), pp.169-177, ACM (Dec. 1981). (Proc. ACM 8th Conf. Operating System Principles, Asilomar, Calif.).

[18] L. A. Rowe and K. P. Birman, "A Local Network Based on the UNIX Operating System.", IEEE Trans. on Software Eng. Vol. SE-8(2), pp.137-146 (Mar 1982).

[19] L. H. Seawright et al, "Papers on Virtual Machine Facility/370", IBM Systems J. Vol. 18(1), pp.4-180 (1979).

[20] M. V. Wilkes and R. M. Needham, "The Cambridge Model Distributed System", Operating System Review Vol. 14(1), pp.21-28, ACM (Jan. 1980).

[21] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal, "The Architecture of the EDEN System", Operating Systems Review Vol. 15(5), pp.148-159, ACM (Dec. 1981). (Proc. ACM 8th Conf. Operating System Principles, Asilomar, Calif.).

[22] B. Goldstein, G. Trivett, and I. Wladawsky-Berger, "Distributed Computing in the Large Systems Environment", Report RC 9027, IBM T. J. Watson Research Center, Yorktown Heights, N.Y. (9 Sept. 1981).

[23] B. Randell, "Recursively Structured Distributed Computing Systems", pp. 3-11 in Proc. 3rd Symp. Reliability on Distributed Software and Database Systems, IEEE (October 1983).

[24] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection - or UNIXes of the World Unite!", Software Practice and Experience Vol. 12(12), pp.1147-1162 (Dec. 1982).