

The Newcastle Connection

or

UNIXes of the World Unite!

D. R. BROWNBRIDGE, L. F. MARSHALL AND B. RANDELL

Computing Laboratory, The University, Newcastle upon Tyne NE1 7RU, England

SUMMARY

In this paper we describe 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 a distributed system, which is currently operational on a set of PDP11s connected by a Cambridge Ring. 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 that we could not resist giving to a software subsystem that we have added to a set of standard UNIX[□] systems in order to connect them together, initially using just a single Cambridge Ring. The resulting distributed system (which in fact can use a variety and multiplicity of both local and wide area networks) is functionally indistinguishable, at both 'shell' command language level and at system call level, from a conventional centralized UNIX system.¹ Thus all issues concerning network protocols and inter-processor communications 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.

This is done, 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 this distributed system, (which for the purposes of this paper we will term UNIX United), the internal design of the Newcastle Connection, the networking and inter-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 in Figure 1.

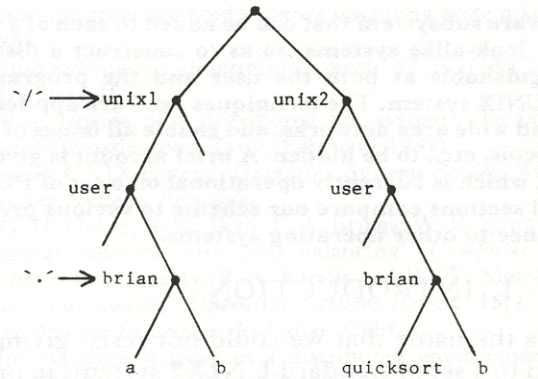


Figure 1

With the root directory (*/*) 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 /../unix/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.)

THE NEWCASTLE CONNECTION

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. (In fact, in our existing implementation, programs are 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.)

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 recognized remote user identifiers, such as 'A/u'. If the system administrator so wishes, rather than refuse all access, he can allow default authentication for unrecognized 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.

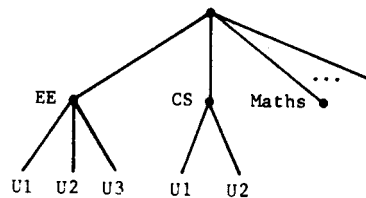


Figure 2

2.2. The structure tree

The naming structure of the UNIX United system represents the way in which the component UNIX systems 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. With the naming structure as shown in Figure 2, 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, owing 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 U.K. country code (44) continues to exist whether or not the transatlantic telephone service is operational.) This issue is pursued further in Section 4 below. One final point: UNIX systems can appear in the naming structure in positions subservient to other UNIX systems. For example, in the previous figure, CS might denote a UNIX system, not just an ordinary directory. This has obvious implications with respect to the respective responsibilities and authority of the various system administrators. If the structure reflects the structure of communication paths, it indicates that all traffic to and from the CS department flows via this particular UNIX system, which is in effect therefore fulfilling a gateway role.

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. This layer of software sits on top of the resident UNIX kernel, i.e. between the UNIX kernel and the rest of the operating system (e.g. shell and the various command programs) and the user programs. From above, the 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 (RPC) protocol,² and is shown schematically in Figure 3.

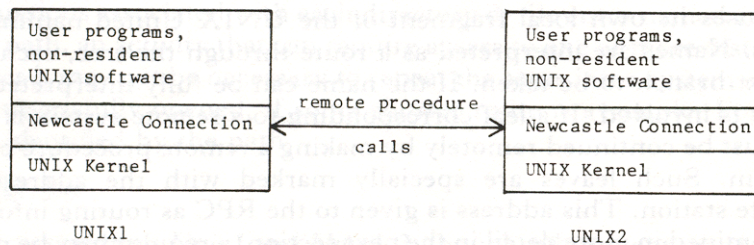


Figure 3

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, the management of alternative routing

THE NEWCASTLE CONNECTION

strategies, etc., are thus all 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. A given UNIX system will itself store only a part of the overall structure. Taking the example given above in Figure 1, *unix1* will store all of the overall structure except those elements that are below *unix2*, and vice versa. Thus, copies of some parts of the structure will be held on several systems, and must of course remain consistent – a problem which would become severe if changes to these parts of the structure were a frequent occurrence.

It is essential for the mapping layer to be able to distinguish local and remote file accesses. The Newcastle Connection layer intercepts all system calls that use files and determines whether the access is local or remote. Local calls are passed unaltered to the underlying local kernel for service; remote calls are packaged with some extra information, such as the current user-id, and passed to a remote machine for service. The Connection uses its own 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 with the address of the appropriate remote station. This address is given to the RPC as routing information. In some cases (examined in more detail in the next section) 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 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 remote 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 file operations. 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 that 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 AND INTER-NETWORKING ISSUES

The various kernels provide mappings between the user-visible name spaces and the hardware name spaces – in particular between the names of what appear to be directories, and the hardware names of distant UNIX systems. It is important that this latter mapping be such that:

- (i) The file naming hierarchy need bear no relationship to the inter-system communications topology

(ii) Modifications to the communications topology should be easy, as well as having no effect on the way in which any user or user program accesses anything.

File naming does, however, have some implications concerning protection and authentication – thus when a path is specified in an open command, checks are made against the permissions associated with each directory or file entry named explicitly or implicitly in the path, an activity that can require access to one or more distant UNIX systems. However it may not be necessary to repeat the same inter-system route when a file has been successfully opened, if some shorter route is available. For example, opening the file identified by the path

././Unix1/a/Unix1.3/b

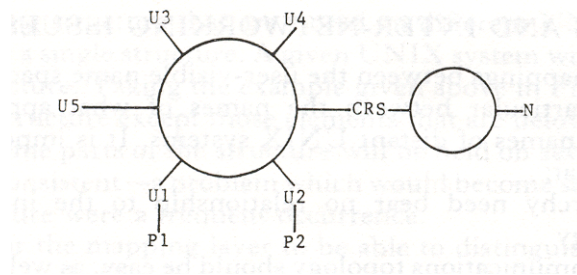
would involve accessing both *Unix1* and *Unix1.3*, though subsequent reads and writes might not involve accessing *Unix1*, depending on the underlying communications topology.

Our approach to providing these facilities takes advantage of the fact that the UNIX and hence the UNIX United file (and device, etc.) naming hierarchy constitutes a set of stable system-wide distinct identifiers. Only one such set is needed, so given that we already have this set available at user level, we are not using the Xerox Ethernet approach, which we understand requires the different stations on a set of interconnected Ethernets to have built-in uniquely assigned identification numbers. Rather, when a UNIX system is introduced into, or physically moved within, a UNIX United system, it will have to be identified to the system using an identifier such as

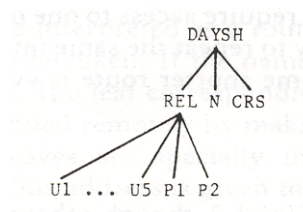
UK/NEWCASTLE/DAYSH/REL/U5

This identification will remain valid no matter the geographical location of the machine.

Our approach also involves arranging that hardware names related to one machine do not permeate to other machines in the system. At the hardware level, each machine identifies the machines it is directly connected to simply by identifying the means of connection, i.e. by I/O device number, and has no means of identifying any other machines. The I/O device numbers remain private to each machine, though presumably all machines on a given network will use the same ring port or modem numbers for a given machine.



(a) Physical Structure



(b) Naming Structure

Figure 4

In Figure 4, the kernel on machine *U1* will make use of hardware addresses for accessing *P1* (identifying the hardwired connection to be used) and *U2*, *U3*, *U4*, *U5* and *CRS* (identifying the ring and the port number to be used). It will not have knowledge of any hardware addresses relevant to *P2* or *N*. Similarly, if it is *U2* that

THE NEWCASTLE CONNECTION

gets connected to a wide area network, say, then only this machine, of the ones shown, will contain any wide area network telephone numbers for other UNIX systems. (Questions of routing through the wide area network will of course not concern the kernel, which only has to deal with routing between UNIX systems, not network nodes.)

Each kernel therefore routes accesses to distant machines to an appropriate one of its adjacent neighbouring machines, which can then pass the request on further, through another wide or local area network, or a direct hard-wired connection, if necessary. (The routing information used is that held in the various special files which of course are accessed by means of the paths or the file descriptors specified by users and user programs.)

Means must be provided for modifying this routing information appropriately when a system is unplugged from one place in the overall network, and replugged into another – or indeed when any changes are made to the communications topology visible at the Newcastle Connection level. (Changes within any of the networks are of no concern, rather the Newcastle Connection has to deal just with changes concerning hardwired machine links and inter-networking.)

We choose to separate the question of updating topological information from that of performance-oriented dynamic routing, and plan to ignore the latter issue (which in any case seems more suitable for networking than inter-networking). In fact some dynamic routing schemes do, almost as a by-product, cope with topological changes, but typically respond rather slowly to such changes.³ A scheme such as that described by Chu,⁴ which arranges the immediate broadcasting of updates to routing tables when a topological change is notified (or perhaps discovered) seems much more appropriate to our needs, and is to be investigated. (Such a distributed approach can be contrasted, for example, with Cambridge's scheme of using a special machine on a ring as a name server, which has to be interrogated whenever an actual ring port number is needed, but thereby simplifies the table updating required when moving a machine from one port to another.)

As a message is passed from one machine to the next, the addressing information it contains (which will be described in terms of paths) will sometimes have to be adjusted, to allow for the fact that movement around the physical structure causes movement around the naming structure. Thus in Figure 4, a message emanating from *P1*, intended for *P2*, will not have its addressing information changed as it travels through *U1* and *U2*, since from both systems, as from *P1*, *P2* would in effect be identified by the path *././P2*. In contrast a message from *U4* to *N* would have its addressing information in effect changed from *./././N* to *././N* as it passed through *CRS*. If system *N* had been made subservient to directory *REL* instead of *DAYSH* then *U4*, having a common parent directory with *N*, would in effect use the path *././N* to send a message to it. En route through *CRS*, this message would need this path to be changed to *././REL/N*, because *CRS* though physically closer to *N* is in fact more distant from it in naming terms. (In practice, file descriptors rather than paths would be used as addressing information – though the principle of changing the addressing information en route still applies.)

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 is 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 economized 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 to embody their solutions in other separate layers of software. Indeed, two significant extensions of UNIX United are already operational, albeit in prototype form. The first of these provides multi-level security, using encryption to enforce security barriers and to control permissible security reclassifications. The second uses file and process replication and majority voting to mask hardware faults – application programs are unchanged, though in fact running in synchronization on several machines with hidden voting. Further details of these and other extensions of the UNIX United scheme are the subject of separate papers.

6. OPERATIONAL EXPERIENCE

At the time of writing (July 1982) the basic UNIX United system is in regular use at Newcastle on a set of three PDP11/23s and two PDP11/45s, connected by a Cambridge Ring. 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 system is now also relied on for network mail, and for solving the problems of overnight file-dumping (of all machines, onto the one tape unit) and of software maintenance and distribution. As mentioned earlier, two prototype extensions of the system, concerned with security and hardware fault tolerance, respectively, are already operational, and work is under way on facilities for replicated files and on a distributed version of MASCOT.⁵

A pre-release version of the Newcastle Connection has been provided to the University of Kent, where work has started on the (it is believed comparatively simple) modifications needed in order to use it to unite several VAX computers, running Berkeley UNIX, also over a Cambridge Ring. The incorporation of X25 network links, and the actual implementation of the mechanisms we have designed for inter-networking, have been delayed by problems beyond our control concerned with the provision of network connections, which it is hoped will be resolved soon.

A first analysis of the performance of the remote procedure call protocol⁶ used by the Connection layer indicates that, subjectively, terminal users of the Newcastle system in general notice little performance difference between local and remote accesses and execution. This is despite the fact that the Cambridge Ring stations used are quite slow, being interrupt-driven devices, and perhaps indicates that such 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. However a further and more extensive programme of performance monitoring and evaluation is planned, which would also include experiments with the DMA ring stations that we have recently obtained and, it is hoped, with more powerful computers than our current set of PDP11s.

The modest size of the Newcastle Connection reflects the need we had to make the system work on our small PDP11/23s, which provided a strong incentive to find 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.) A program that made use of every facility provided by the Newcastle Connection would be increased in size by slightly more than 7K; however, it is very unlikely that this would ever be the case and a more usual figure would be an increase of 4'5K, the absolute minimum being 3K (including the code for the RPC interface).

THE NEWCASTLE CONNECTION

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 interconnected 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 System⁷ does for users of various transaction processing programs, and what the National Software Works project⁸ 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 specialized 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 realization 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' facilities¹⁰ 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 appear 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 Network¹³ 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 servers¹⁴ goes further by providing each workstation user with an 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 RIDE¹⁵ 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 paper¹⁷ 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 resting 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 reinvention.) In fact we were not consciously aware of any of these systems (other than ‘uucp’ and of course DRFS) while 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, 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 specialized service protocol.^{20, 21}

Returning to the layered (‘level of abstraction’) aspect of UNIX United and its various extensions, this of course can be seen as being directly in the tradition pioneered by Dijkstra’s seminal THE operating system²² – a system which it now seems to be as unfashionable to reference as it was once fashionable. However, if only through our extensive work on multi-level structuring for purposes of fault tolerance,²³ we remain convinced that this approach to designing (and describing) systems is of great merit. It leads to designs which work well and which seem to us to be much easier to comprehend, and therefore have faith in, than those where little explicit thought has apparently been given to achieving any sort of separation of logical concerns, such as those of naming, communications routing, load balancing, recoverability, etc.

THE NEWCASTLE CONNECTION

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 principle 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 organized 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 virtualizing 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 reasonably satisfactory high level language. Our method of incorporating the Newcastle Connection 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.*²⁴ 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 userid*. 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.

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 – if 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 realize. 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. 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.

ACKNOWLEDGEMENTS

Many of our colleagues at Newcastle have had a part in the often apparently furious, yet always enjoyable, discussions out of which the Newcastle Connection scheme, and its internal design, evolved. Two colleagues in particular, Fabio Panzieri and Santosh Shrivastava, also deserve acknowledgements as designers and implementors of the Remote Procedure Call protocol on which we have relied so heavily. The X25 protocol software that we are planning to use was provided by Keith Ruttle of the University of York, who is being most co-operative with our efforts at burying his software in a way that he could hardly have foreseen or intended.

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, we are pleased to acknowledge that our work has been supported by research contracts from the U.K. 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*, **17** (7), 365-375 (1974).
2. S. K. Shrivastava and F. Panzieri, ‘The design of a reliable remote procedure call mechanism’, *IEEE Trans. Computers*, **C-31**, (7), 692-697 (1982).
3. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, N. J. (1981).
4. K. Chu, ‘A distributed protocol for updating network topology information’, *Report RC 7235*, IBM T. J. Watson Research Center, Yorktown Heights, New York (27 July 1978).
5. *The Official Handbook of MASCOT*, MASCOT Suppliers Association (5 December 1980).
6. F. Panzieri and S. K. Shrivastava, ‘Reliable remote calls for distributed UNIX: an implementation study’, *Report 177*, Computing Laboratory, University of Newcastle upon Tyne (June 1982).
7. J. Gray, ‘IBM’s customer information control system (CICS)’, *Operating System Review*, **15** (3), 11-12, (1981).
8. R. E. Millstein, ‘The national software works: a distributed processing system’, *Proc. ACM 1977 Annual Conference*, Seattle, Washington, 44-52 (1977).
9. R. Rashid, ‘Accent: a communication oriented network operating system kernel’, *Operating Systems Review*, **15** (5), 64-75 (1981).
10. D. A. Nowitz, ‘Uucp implementation description’, Sect. 37 in *UNIX Programmer’s Manual, Seventh Edition, Vol. 2* (January 1979).

THE NEWCASTLE CONNECTION

11. M. Jegado, 'Recoverability aspects of a distributed file system', *Technical Report*, Computing Laboratory, University of Newcastle upon Tyne (February 1981).
12. G. L. Chesson, 'The network UNIX system', *Operating Systems Review*, **9** (5), 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*, **15** (4), 55-66 (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', *Operating Systems Review*, **15** (5), 160-168 (1981). (*Proc. ACM 8th Conf. Operating System Principles*, Asilomar, Calif.).
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 December 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*, **15** (5), 169-177 (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. Software Eng.*, **SE-8** (2), 137-146 (1982).
19. L. H. Seawright *et al.*, 'Papers on virtual machine facility/370', *IBM Systems J.*, **18** (1), 4-180 (1979).
20. M. V. Wilkes and R. M. Needham, 'The Cambridge model distributed system', *Operating System Review*, **14** (1), 21-28 (1980).
21. E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler and S. Vestal, 'The architecture of the EDEN system', *Operating Systems Review*, **15** (5), 148-159 (1981) (*Proc. ACM 8th Conf. Operating System Principles*, Asilomar, Calif.).
22. E. W. Dijkstra, 'The structure of the "THE" multiprogramming system', *Communications of the ACM*, **11** (5), 683-696 (1968).
23. T. Anderson, P. A. Lee and S. K. Shrivastava, 'A model of recoverability in multi-level systems', *IEEE Transactions on Software Engineering*, **SE-4** (6), 486-494 (1978) (also *Report 115*, Computing Laboratory, University of Newcastle upon Tyne).
24. 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, New York (9 September 1981).