Abstract:

Two recursive structuring principles which aid the construction of sophisticated distributed computing systems are described, namely that:

(i) a distributed computing system should be functionally equivalent to the individual computing systems of which it is composed, even with respect to exception reporting (i.e. the information that a system provides to its environment when it is unable, or perhaps even not designed, to carry out a requested operation), and

(ii) fault tolerant systems should be constructed from generalised fault tolerant components. (Such components try to tolerate, wherever appropriate, their own faults and those reported to them by underlying components, and also being wrongly invoked - both by the components they interact with and the component of which they themselves form part.)

The first principle motivates the use of a strict context-relative naming scheme for all objects in the computing system - taken together the principles enable the problems of constructing coherent systems from heterogeneous components, of incorporating fault tolerance and of providing multi-level security, to be greatly simplified by being treated as essentially separable logical problems.

An operational distributed computing system based on UNIX* and designed in accordance with these principles is used for illustration. This system has been implemented by adding a software subsystem, known as the Newcastle Connection, to each of a set of UNIX systems, so as to construct a distributed system which is functionally equivalent at both the user and the program level to a conventional uniprocessor UNIX system. Prototype extensions of the system, providing multi-level security and hardware fault tolerance, have also been produced, and are briefly described, as are plans to incorporate non-UNIX systems into the overall distributed system.

*UNIX is a Trademark of Bell Laboratories.

The Structuring of Distributed Computing Systems

By

B. Randell

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 181
December, 1982

© 1982 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
have also been produced, and are briefly described, as are plans to incorporate non-UNIX systems into the overall distributed system.

1. INTRODUCTION

Careful structuring is of course crucial to the success of any complex computing system design. Appropriately chosen internal interfaces and system components can make the overall system much easier to comprehend, and hence to construct, validate and, if necessary, to modify. This applies whether one is concerned with the design of a large software system, resident on a single computer, or the software and hardware making up a distributed computing system.

In this latter case, allocation of separate software functions to separate computers can make the overall structure of the system much more explicit than when all the software is held in the memory of, and executed by, a single computer. Thus some designers of distributed computing systems have taken as their main structuring principle the identification of functions that can be so treated and their implementation as so-called "servers" - name servers, file servers, boot servers, mail servers, compiler servers, server servers, etc. However, as we will seek to show, there is much more to the topic of structuring distributed computing systems than this. Indeed such an approach, unless carried out in accordance with an appropriate overall system architecture, can lead to systems which are difficult to modify or extend, for example in response to changed workloads.

In this paper various structuring principles and techniques, some but not all of them reasonably well known, are discussed not just in abstract terms but rather as they relate to a distributed system that we have designed and implemented at Newcastle. Our work has in fact involved the development of a software sub-system (called the Newcastle Connection) which can be added to each of a set of physically connected UNIX or UNIX-lookalike systems in order to turn them into a distributed system (which for the purposes of this paper will be called a UNIX United system). Such a system has been operational at Newcastle for some months on a set of PDP11s connected by a Cambridge Ring; pre-release versions of the Connection subsystem have been made available to a small number of other organisations for experiments using various computers and versions of UNIX. Full details of UNIX United and the Newcastle Connection can be found in Brownbridge et al [1], which also surveys a number of related systems developed elsewhere.

The discussion of structuring techniques forms the subject of the next two sections of this paper, with Section 3 also including an overview of the structure of the UNIX United system. These sections provide a logical framework which allows the topics of distributedness, heterogeneity, fault tolerance and security to be treated as essentially separate design issues. - in fact as the subjects of Sections 4 to 7, respectively, with Section 8 containing brief concluding remarks.

December 3, 1982
2. BASIC STRUCTURING TECHNIQUES

One common form of system structuring is that of division of software and/or hardware into (or equivalently, construction of a system out of) a set of co-existing interacting components. Such structuring only makes sense if the interfaces between components are, so to speak, "narrow". Such interfaces are ones which

(i) can be specified more simply than the internal construction of the components that they separate can be described (e.g. procedures with a small number of reasonably simple parameters, or hardware components with relatively few connecting wires), and

(ii) are placed across low bandwidth information transmission paths.

This form of structuring is more easily described than applied, since the correct choice of interfaces (presuming it is not decreed by other factors, such as the requirement to use given pre-existing components) often requires considerable experience and insight. For example, distributed systems that have been structured into multiple specialised servers, but with badly positioned interfaces, could be unnecessarily complicated and/or have very poor performance characteristics indeed, with much time lost moving large amounts of information from computer to computer, or waiting in line for particular heavily congested servers.

The technique of structuring a system by dividing it into a set of co-existing interacting components can be contrasted with the technique which is associated with the phrase "level of abstraction", in which one or more components are constructed using ("on top of") other components. This second form of structuring is commonly thought of, at least in software, in linguistic terms, with an interface defining a language of operations, data types, etc., which the lower level implements and the upper level is programmed in. Perhaps the most notable early application of this program structuring technique to the construction of an operating system was Dijkstra's TIME multiprogramming system [2]. In this system a set of conceptually separate levels of abstraction dealt with a set of separate issues, such as processor scheduling and memory allocation. Although this system was a multiprogramming system, the structuring concepts it embodies are as relevant to distributed systems as to centralised systems, and are used extensively in UNIX United.

3. THE PRINCIPLE OF RECURSIVE STRUCTURING

The single most important structuring principle that is reflected in the design of UNIX United is that:

A distributed computing system should be functionally equivalent to the individual computing systems of which it is composed, even with respect to exception reporting (i.e. the information that a system provides to its environment when it is unable, or perhaps even not designed, to carry out a requested operation).

December 3, 1982
This recursive structuring principle has previously been advocated and investigated, both at Newcastle and elsewhere, in connection with computer architectures intended specifically for VLSI implementation [3,4]. The principle aim was to design a processor architecture which need not be affected by changes in level of integration - to get away from the apparently inexorable progression of incompatible microprocessor architectures, with ever bigger word lengths and instruction sets, that have accompanied the evolution of VLSI technology. Instead, with a "recursive" architecture, as increased integration levels make it possible, an ever greater number of component processors are fitted within a single chip, so improving its performance without affecting its functionality.

To the best of our knowledge, UNIX United is among the first examples, if it is not the first example, of the application of this principle to operating systems. It is perhaps worth remarking that in computing systems design an architectural concept of any real merit should at least in principle be equally applicable at any of a number of levels, e.g. microprogram, program, operating system, database access, etc.

Adherence in the design of a distributed system to the recursive structuring principle provides, at the level of complete computing systems, the sort of benefits that the concept of block structure in Algol 60 provided to programmers over what had been available hitherto, in languages such as FORTRAN and COBOL. For example, it makes possible an extremely simple means of joining existing distributed systems together - something that would not be at all easy with, say, the Cambridge Model Distributed System [5], because of the sort of naming and addressing mechanisms it incorporates. Equally, and perhaps less obviously, adherence to the principle facilitates the partitioning of a system into separate pieces, a subject to which we will return in Section 7 in connection with the problem of providing multi-level security.

In essence, the value of the principle is that - by definition - a distributed system which is recursively structured in this way is indefinitely extensible, at least in theory. Indeed UNIX United has been designed with the intention of constructing a very large distributed system, involving both wide and local area networks.

The component systems of which a recursively structured distributed system is constructed must, on the other hand, possess characteristics that are appropriate for the distributed system as a whole - firstly, they must provide (at least the appearance of) parallel processing facilities. This even a uni-processor UNIX does, because of its ability to allow users and their programs to initiate asynchronous processes. In a UNIX United system such processes may, without having to be changed in any way, in fact be run on separate processors, so that quasi-parallelism is transformed into actual parallelism.

Secondly, the various objects within a system (computers, data items, I/O devices, etc.) must be accessible by means which are independent of whether the system is in fact a complete one, or merely a component of a larger system. Thus component computers should support a general "contextual naming" scheme for their various objects. In other

December 3, 1982
words, there should be means for introducing and entering (and leaving) new naming contexts, and all names must be context-relative. This is a characteristic that UNIX possesses by virtue of its hierarchical scheme for naming files, devices and commands, in which directories serve as the required contexts.

```
root `/' -->
\ / \ /  
user 1ib
\ / \ /
current working --> brian fred
\ / \ /
directory dirl b c
\ / \ /
a b
```

Figure 1: The UNIX Name Space

Figure 1 shows part of a typical UNIX naming hierarchy. Files, directories, etc., can only be named relative to either the directory which is designated as being the "current working directory" or that which is designated as the "root directory". Thus "/user/brian/dirl/a" and "dirl/a" identify the same file, the convention being that a name starting with "/" is relative to the root directory. Objects outside a context can be named relative to that context using the ".." convention to indicate a parent directory. (Note that this avoids having to know the name by which the context is known in its surrounding context.) The names "/user/fred/b" and "./.fred/b" therefore identify the same file, the second form being a name given implicitly relative to the current working directory rather than the root directory.

As its name implies, the current working directory can be changed. In fact the root directory can also be re-positioned. In both cases however, this can be done only by specifying a context-relative name. There is on the other hand no means of specifying an absolute name, relative to the base of the tree, say. The base directory, which is usually but not necessarily chosen as the root directory, can itself be recognised only by the convention that it is its own parent. Moreover all other means provided for identifying any of the various kinds of objects that UNIX deals with, e.g. users, processes, open files, etc., are related back to its contextual naming scheme.

This simple and elegant scheme of context-relative naming has been taken advantage of in UNIX United by identifying individual component UNIX systems with directories in a larger name space, covering the UNIX United system as a whole. Any directory can be associated with a

December 3, 1982
separate UNIX machine - in Figure 2 we show how a UNIX United system spanning an entire university might be created from the machines in various university departments, using a naming structure which matches the departmental structure (without regard to the actual topology of the underlying communications networks).

![Diagram of a university-wide UNIX United System]

Figure 2: A University-Wide UNIX United System

The figure implies that from within the Computing Science Department's U1 machine, files on its U2 machine will normally have names starting "/.../U2" and files on the machine that the Electrical Engineering Department has also chosen to call "U2" will need to be identified with names starting "/.../EE/U2". (UNIX has various means for, in effect, abbreviating lengthy names, which need not concern us here.)

By taking advantage of UNIX's contextual naming scheme in this way, it has been possible to produce a distributed system which is functionally equivalent at both the user and the program level to a conventional uni-processor UNIX system, even with respect to error reports. All the standard UNIX conventions, e.g. for protecting, naming and accessing files and devices, for input/output redirection, for inter-process communication, etc., are applicable without apparent change to the system as a whole. All issues of inter-processor communication, networking protocols, etc., are hidden. Thus the standard UNIX facilities can be used without concern for the fact that several machines may be involved, and the user need have no knowledge of what data and messages flow when, or between which machines, or which processor actually executes any particular program.

In fact the present implementation of UNIX United runs programs on the machine within whose file store their program code is held. Thus the command line

December 3, 1982
which might have been entered on a terminal logged in to U4, say, causes
the "sort" program to be run on machine U1, with its data being fetched
from U2 and the results being piped to the "summarise" program running
on U3 in parallel with "sort". On the other hand if U1 and U3 were
merely directories on a single UNIX machine, "sort" and "summarise"
would have run in an interleaved fashion.

Adoption of the recursive structuring principle thus has a profound
(and highly beneficial) effect on the usability of a distributed system.
It also provides, in conjunction with the principle of the separation of
logical concerns, a number of valuable guidelines as to how to tackle
the various implementation issues, including the provision of fault
tolerance and multi-level security, and also the construction of a
coherent system from a collection of heterogeneous components.

4. DISTRIBUTEDNESS

In our view, the structuring principles discussed above make it
particularly natural to regard "distributedness" (i.e. the fact that a
system incorporates a set of autonomous yet interacting computers) as
providing a design problem which is clearly separable from various other
issues, most notably fault tolerance, with which many papers and designs
often link it apparently inextricably, e.g. Popek [6]. We regard the
problem as involving two principal issues, namely those of:

(i) routing each request for activity to the appropriate component
computing system, from the one in which it originated, and

(ii) preserving the appearance of a single overall recursively struc-
tured name space, and hiding any local addressing mechanisms used
inside the component computing systems in support of this name
space.

The provision of atomic actions, for example, is regarded as a
separate issue (addressed in Section 6), since such facilities are at
least in principle of as much relevance to a computing system which
merely provides the appearance of parallel (and hence possibly interfer-
ing) processes using a single processor as one in which there is actual
parallelism. In other words, atomic actions are as relevant to a time
sharing system as to a distributed system.

We have accordingly tried to identify the minimum set of facilities
that are needed for the provision of distributedness (in a recursively
structured system) and to implement them in a clearly separate mechan-
ism. UNIX United has in fact been implemented merely by adding a sub-
system, in the form of a software layer, to an otherwise unchanged UNIX
system. In direct analogy with the THE hierarchy of levels of abstrac-
tion, other equally strictly distinguished software layers, relevant to
other separable logical concerns, can be placed above (or below) the
Newcastle Connection layer, as appropriate.

December 3, 1982
The positioning of the Connection layer is governed by the structure of UNIX itself. In UNIX all user processes and many operating system facilities (such as the 'shell' command language interpreter) are run as separate time-shared processes, able to interact with each other, and the outside world, only by means of 'system calls' -- effectively procedure calls on the resident nucleus of the operating system, the UNIX kernel. The Connection is a transparent layer that is inserted between the kernel and the processes. It is transparent in the sense that from above it is functionally indistinguishable from the kernel and from below it appears to be a set of normal user processes. It filters out system calls that have to be re-directed to another UNIX system (for example, because they concern files or devices on that system), and accepts calls that have been directed to it from other systems. Thus processes on different UNIX machines can interact in exactly the same way as processes on a single machine. (There is a quite separate issue of whether UNIX needs additional forms of inter-process communication, e.g. for synchronous message passing between unrelated processes. Clearly, if such facilities were added to the UNIX kernel the Connection layer would have to be extended in order to ensure that they worked for processes that happened to be on different machines.)

Since system calls act like procedure calls, communication between the Connection layers on the various systems is based on the use of a remote procedure call protocol [7], and is shown schematically below.

```
User programs,    | User programs,
non-resident      | non-resident
UNIX software     | UNIX software

--------------------  ---------------
Newcastle Connection|  ---------------
calls                | Newcastle Connection
--------------------  ---------------
UNIX Kernel          | UNIX Kernel
```

**Figure 3: The Position of the Connection Layer**

A slightly more detailed picture of the structure of the system would reveal that communications actually occur at the hardware level, and that the kernel includes means for handling low level communications protocols. However all such issues are hidden from the user of UNIX United.

It is of course still left to each UNIX programmer to choose to implement a given algorithm in the form of a single process, or alternatively as a set of interacting processes, so as to take advantage of the quasi-parallelism in UNIX, and perhaps real parallelism in UNIX United. Thus the existence of the Newcastle Connection still leaves open the question of whether a centralised or (logically) distributed

December 3, 1982
implementation of, say, a data base manager is most appropriate in given circumstances - physical distribution has however been subsumed within logical distribution.

4.1. Names and Addresses

In a recursively structured system each component computer possesses what appears to be a complete name space, but which in fact is just part of the overall name space. Thus one of the consequences of distributedness is the requirement for some means of combining these component name spaces.

The technique that we have evolved for this purpose in UNIX United is as follows. Each component UNIX system stores just a part of the overall naming structure. In fact each system stores the representation of its own section of the naming tree. However each system also stores a copy of those parts of the overall naming structure that relate it to those 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).

![Diagram of the Name Space]

Figure 4: Representing the Name Space

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

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 various 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 since names stored in files or incorporated in programs (or even just known to users) may be invalidated.

December 3, 1982
It is not reasonable to assume that all accessing of all types of object in a distributed system will be performed using general contextual names each and every time. Rather, in the interests of efficiency, and in order to exploit actual hardware provisions, names will when appropriate be bound to what are effectively addresses, of limited scope and validity, so that repeated accesses to a given object can then be made by just using its address.

For example, in UNIX the act of opening a file (identified by a contextual name) causes what is effectively an address, local to the current process, to be assigned for use in subsequent accesses to that file. This address (a UNIX "file descriptor") will be valid only until the file is closed. If the file is to be accessed by a process on a remote system, a file descriptor which is valid on that system must be used. However this file descriptor must be associated with the relevant file descriptor on the system holding the file. In UNIX United this task is carried out within the Connection layer, using appropriate mapping tables. (There is of course more to the subject of names and addresses than this - see for example Saltzer[8]. Indeed the facilities in UNIX for name binding have some inadequacies. However the job of the Connection layer is to produce a fully compatible distributed version of UNIX, and not to "improve" its functionality in any way.)

4.2. And As For Name Servers ..

Given the comments in the Introduction about structuring a distributed system out of specialised servers on separate computers it is perhaps worth describing explicitly how one such server concept, that of a "name server", fits into the UNIX United scheme. The basic function of a name server is to provide a central repository for information regarding the physical addresses of the various other components of the distributed system, information that can then be used to enable these components to be accessed directly.

A UNIX United system can easily be set up, using a Ring network, say, to work in just this way, as is illustrated in Figure 5.

![Figure 5: A Name Server](image-url)

December 3, 1982
Here all but one of the component UNIX systems are made subservient, in terms of the global naming tree, to the one other system, labelled NS in the diagrams. This system will contain hardware addresses (ring port numbers) for all of the others, each of which will hold the hardware address of just the NS system. If U1, say, needs to access a file on U2 it can "open" the file using a name which starts "/../U2" — hence the name of this section! The "open" system call will have to access NS in order to check permissions, but will in due course return the port number of U2 so that thereafter reads and writes to the file will go directly to U2, and not involve NS.

The name server idea is thus seen as just a specialised usage of the general UNIX United scheme, and in particular of the route optimisation that the Connection implements amongst a set of UNIX systems that are all in the same hardware address space, be it a Ring, an Ethernet, an X25 network, or whatever.

5. HETEROGENEITY

Much work on distributed systems, particularly on network protocols, mixes up two separate issues, namely distributedness and heterogeneity. Thus typical file transfer protocols, virtual terminal protocols, etc., are as much concerned with incompatibilities between computing systems as the fact that several computing systems are involved. Our approach, based on isolating and then separately solving the problem of distributedness, would appear to be of relevance solely to computing systems composed of identical component computing systems. This is not the case — the components just have to be functionally equivalent to each other, a task that is made easier by the fact that there are defined ways of responding to system calls which cannot be carried out, for whatever reason.

For example, one could connect together UNIX 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 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. (In fact when a UNIX United system is implemented using a number of different types of computer, low level incompatibilities such as differing number representations might not be fully hidden from users.)
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 particular it can be seen as unifying the two apparently distinct forms of structuring discussed in Section 2, since essentially the same interface is used both to co-existing and to underlying components.

In fact one example of the use of specialised components was mentioned earlier, i.e. the name server. However, though the name server was discussed as though it was a standard UNIX system (perhaps even with its own files and processing activities) this does not have to be the case. Rather, if it is functioning solely as a name server it could well make sense for it to have been implemented specially, and not based on a UNIX kernel, so long as it responds properly, if only with exception messages, to the various UNIX system calls.

Another specialised component that is being investigated at Newcastle is a terminal concentrator. The concentrator is designed to serve as part of an existing campus network, serving various sorts of host computing systems, and is in no way related to UNIX. However it is now being extended so as to have a (very limited) remote UNIX system call interface, so that it can be linked to a UNIX United system, from which it will appear to be a conventional UNIX system whose naming tree contains just terminals.

Another development that is being considered is that of providing a limited remote UNIX system call interface on a totally different operating system. Initially just the basic system calls concerned with file accessing would be supported, and mapped into equivalent facilities within this other system. The simplicity and extensibility of this approach contrast favourably with the more conventional one of having each operating system support a general file transfer protocol, particularly since it enables a remote file to be accessed and updated selectively.

Perhaps the most ambitious approach to the problems of linking heterogeneous computing systems together is the Open Systems Interconnection (OSI) scheme [9]. This involves the creation and definition of what are hoped will prove widely acceptable abstractions of a number of concepts that exist in differing forms in different systems, such as files, processes and transactions. In fact the OSI scheme can be viewed as an attempt to specify an abstract operating system, unfortunately one whose adequacy and merits cannot be known until its functions have been mapped successfully on to one or more probably unco-genial host operating systems. In comparison our UNIX-based approach provides a form of "coherent heterogeneity" based on a proven set of abstractions, already successfully implemented on a wide and ever-growing variety of different

* LEGO is a Registered Trademark of LEGO Systems A/S.

December 3, 1982
hardware. Moreover it is our belief that the UNIX system calls are sufficiently simple, yet general, to be used as a common basis on to which one should be able to map many of the facilities of a wide variety of different operating systems. Thus in relation to OSI's famous seven levels of protocols, one could say that UNIX United hides the bottom four levels and in some senses does away with the need for the other three, substituting for them the full UNIX system call interface.

6. FAULT TOLERANCE

System structuring techniques play a very large role in the provision of effective fault tolerance, as is discussed at length by Anderson and Lee [10]. This describes our (of course) recursive approach to the construction of fault tolerant systems out of fault tolerant components, such that, at least in principle, each level of component can contain facilities for trying to tolerate:

(i) faults in underlying components that are reported to it,
(ii) its own faults, and
(iii) faulty invocation of the component by its environment, i.e. the enclosing component, or a co-existing component with which it is interacting.

Compared to a centralized system, a distributed system provides new opportunities for the provision of high reliability by means of fault tolerance, and also new types of fault that could impair reliability unless properly tolerated. These are logically separable issues and should be treated as such. Moreover they are also separable from any opportunities or requirements for fault tolerance that would exist in an equivalent non-distributed system.

In line with this attitude regarding the separation of logical concerns, UNIX United is structured so that the only reliability problems which are treated within the Connection layer are those which arise specifically from the fact that the system is distributed. Nevertheless the internal structure of the Connection layer is itself worthy of note.

The Connection layer uses a remote procedure call (rpc) protocol which addresses the problems caused by breakdowns ("crashes") of the component computers and communication links, and the occasional loss of messages across the links. In these circumstances it could be all too easy for a remote procedure to be accidentally executed several times - our rpc protocol attempts to prevent this and to achieve an "exactly once" semantics [11]. There remain the problems of the computer which is trying to make a (perhaps related) series of remote procedure calls itself crashing on occasion, or of the Connection layer, despite its best efforts, being unable to achieve all the requested calls. These problems are not regarded as the province of the Connection layer. Rather they are essentially similar to problems that can arise in a centralized (multi-programming or time-sharing) system - problems for which recoverable atomic actions (ones which are guaranteed either to succeed or to do nothing) provide an answer. (All of these issues are treated in

December 3, 1982
much greater detail in papers relating to the rpc protocol.) [12,13]

Though it follows from application of the recursive structuring principle, it is worth mentioning explicitly that the Connection layer reports any errors that it cannot recover from in terms similar to those used by the UNIX kernel. Thus it reports merely that a file cannot be opened, rather than, say, that the communications line to the machine containing the file is not operational. (In practice, facilities to aid fault location and repair may well be needed, but this is regarded as a separate issue from that of exception reporting for purposes of fault tolerance.)

Support for atomic actions per se is not regarded as part of UNIX United, since it would augment the functionality of UNIX itself. However an appropriate further software layer is being developed which will provide UNIX, and hence UNIX United, with atomic actions which are recoverable, at least with respect to file usage. It is based on the Distributed Recoverable File System [14] developed earlier at Newcastle for UNIX. In essence the new layer will just provide three additional system calls:

(i) Establish Recovery Point (i.e. start state-saving, and locking files),

(ii) Discard Recovery Point (i.e. discard saved state, and unlock relevant files), and

(iii) Restore Recovery Point (i.e. go back to latest uncommitted recovery point).

Adding this layer to the system is best done by placing it between the kernel and the Connection layer, whose provisions will therefore have to be augmented to deal with the three additional system calls. (In the case of the Discard Recovery Point call, it might well be thought necessary to incorporate a simplified form of "two-phase commit protocol" [15], which would involve the provision of another system call "Prepare to Discard Recovery Point" by the Atomic Action layer. This should minimise the risk of having some but not all the component UNIX systems complete their Discard Recovery Point calls.) The resulting software structure for each component computer will be as shown in Figure 6 below:

December 3, 1982
Figure 6: Fault Tolerance Layers

Another well known form of fault tolerance is that of using replication and majority voting, typically with the aim of masking operational hardware faults. A prototype extension to UNIX United has already been constructed which uses this approach. It has involved adding an additional transparent software sub-system (the Modular Redundancy layer) to each of a number of UNIX systems on top of their Connection layers. Copies of a conventional application program and its files can then be loaded onto each of three systems and run so that file accesses are synchronized and voted upon. Any malfunctioning computer so identified by the voting is automatically switched out and in due course another switched in to replace it.

This of course is not a new idea - a well-known computing system using this technique is the SIFT system [16]. The point is that the technique is very simple to implement when it is separated from issues of distributedness. Needless to say, given that the Modular Redundancy layer is transparent, one can envisage using both it and the Atomic Action layer together, the latter having the task of trying to cope with situations where the problem is not a hardware fault, but one arising, say, from erroneous input data.

The significance of the simplicity, generality and mutual independence of these various mechanisms when used in conjunction with the notion of recursive structuring is considerable. Complexity is one of the major impediments to reliability, so that complicated and needlessly interdependent fault tolerance mechanisms are more likely to reduce than to improve reliability, because of the danger of situations arising, particularly during error recovery and system reconfiguration, that have not been catered for properly.

December 3, 1982
7. SECURITY

The recursive structuring principle implies that the security provisions existing in component computing systems must be mirrored exactly by the distributed system as a whole. In fact 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. Any Connection layer, when receiving messages containing system calls diverted to it by the Connection layer on some other system, then only needs to authenticate the identity of the system which is the source of its messages.

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 "login". 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.

Though the standard UNIX security facilities, when carefully used, are as good if not better than those of most other time-sharing systems, they are quite inadequate against determined attack by would-be penetrators. There are many situations where the potential costs of a security violation, and the danger of deliberate attempts to subvert the security controls, are so great that positive assurances regarding the security of a system are required. (This is particularly the case if a multi-level security policy is involved.) It then does not suffice merely to have "plugged" all the various gaps that previous penetrations have revealed. Rather, a compelling argument guaranteeing that the system has been designed and implemented so as to adhere to the required security policy must be supplied. In these circumstances it is essential to construct the system in such a way that the mechanisms on which security depends are clearly identified, and simple enough for their adequacy to be manifest - ideally for their correctness, with respect to some formal statement of the required security policy, to be proven formally, perhaps by, or with the aid of, a program verification system.

December 3, 1982
Two important types of security mechanism are those that prevent information flow, and those that monitor and mediate such information flow as is allowed, between system components that cannot be trusted to adhere to the security policy, perhaps because of their complexity. The recursive structuring technique can readily be used to construct a system that provides multi-level security, by allocating separate (untrusted) general-purpose component computers to different security levels and implementing appropriate (trusted) security mechanisms as transparent additions to the inter-processor communication links. A prototype of such a system has in fact already been constructed (indeed in just a few days) based on UNIX United, using encryption-based security mechanisms (called "Z-boxes") to prevent information flow between security regimes, and to control the types of security reclassification allowed. This system is portrayed in Figure 7 above.

Construction of a much more sophisticated version of this system is now planned, a description of which is given in Rushby and Randell [17]. The system structuring techniques involved enable us to have some confidence that the design of the security-relevant aspects of the system will be simple enough to be amenable to formal specification and verification. There is also every reason to believe that the system will not suffer the sort of severe performance degradation that has resulted when attempts have been made to provide multi-level security in a general purpose operating system running on a single computer. Furthermore, there should be nothing to prevent the immediate incorporation of the sort of fault tolerance mechanisms described earlier, without impacting

December 3, 1982
the trustworthiness of the security mechanisms, due to the careful separation of the various issues involved.

8. CONCLUSIONS

In our view, the ideas that we have tried to encapsulate in the form of two "structuring principles" provide a surprisingly effective and constructive methodology for the design of distributed systems, and for dealing with issues such as reliability and security. Moreover they give a means of making a coherent system out of a heterogeneous set of components - in part by hiding the heterogeneity behind a facade of homogeneity, in part by treating it as exceptional behaviour for which fault tolerance provisions can be made.

Certainly our experience with UNIX United provides what we regard as strong evidence for the merits of this methodology. As reported in [1], a very useful distributed system, enabling full remote file and device access, was constructed within a month of starting implementation of the Connection layer. Needless to say, the fact that - due to the transparency of the Newcastle Connection - it was not necessary to modify or in most cases even understand any existing operating system or user program source code was a great help! In a very few months this system had been extended to cover remote execution, multiple sets of users, etc., two prototype extensions of the system, for multi-level security and hardware fault tolerance, had been successfully demonstrated, and the design of others commenced. However we still feel we have barely begun to explore all the many possible ramifications of the scheme, and of course there are many evaluation exercises and engineering improvements to be investigated.

What has been presented here as a discussion of structuring principles for the design of distributed computing systems could equally well be viewed as a rationale for the design of UNIX United. It would be nice to be able to report that the process of designing UNIX United had been guided, at all times, by explicit recognition of these principles. In practice the above account is as much a rationalisation of, as it is a rationale for, the design of UNIX United. The various structuring ideas, in particular those on fault tolerant components and on recursive architectures, had already been a subject of much work at Newcastle, but the work that led to the specification and detailed design of the Newcastle Connection has contributed to, as well as greatly benefitted from, our understanding of system structuring issues. Equally it owes much to the external form (if not internal design) of UNIX itself - the only operating system we know of which is at all close to being an ideal component of a recursively structured distributed computing system. Nevertheless we would not wish to give the impression that UNIX is perfect, and that these structuring ideas are relevant only to UNIX and UNIX-like systems. Rather, we believe that the ideas provide an interesting perspective from which to judge the merits of existing or planned systems, and can help to guide further work on the design of operating systems and distributed systems.

December 3, 1982
9. ACKNOWLEDGEMENTS

The discussion of structuring issues presented in this paper owes much to the work of the author's many colleagues, at Newcastle and elsewhere, over a number of years. Recent work specifically on the Newcastle Connection, however, has been mainly in close collaboration with Lindsay Marshall, Dave Brownbridge, Fabio Panzieri and Jay Black. The work on multi-level security has been largely carried out with John Rushby, and that on hardware fault tolerance with Li-Yi Lu. Research on distributed computing systems at Newcastle is sponsored by the U.K. Science and Engineering Research Council and the Royal Signals and Radar Research Establishment.

December 3, 1982
REFERENCES


December 3, 1982


December 3, 1982