System design and structuring for dependability

B. Randell

TECHNICAL REPORT SERIES
No 232 February, 1987
System design and structuring for dependability

B. Randell

Abstract

The complexity of a large and sophisticated computing system can be reduced significantly by ensuring that the system is constructed out of a well-chosen set of largely independent components, whose normal and exceptional interactions are properly defined and well-understood. However the task of structuring a system, i.e. of choosing and defining appropriate components, can be very difficult. This paper describes a technique of system structuring which involves distinguishing the functionality which a system is intended to have from other desirable attributes, including such aspects of dependability as reliability and security. Separate components are then used to provide each of these attributes, in the framework of a hierarchical exception handling scheme. Various UNIX-based components which have been implemented at Newcastle are used to illustrate this approach.


Series Editor: M.J. Elphick

© 1987 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.
Bibliographical details

RANDELL, Brian

System design and structuring for dependability
[By] B. Randell


(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 232.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

The complexity of a large and sophisticated computing system can be reduced significantly by ensuring that the system is constructed out of a well-chosen set of largely independent components, whose normal and exceptional interactions are properly defined and well-understood. However the task of structuring a system, i.e. of choosing and defining appropriate components, can be very difficult. This paper describes a technique of system structuring which involves distinguishing the functionality which a system is intended to have from other desirable attributes, including such aspects of dependability as reliability and security. Separate components are then used to provide each of these attributes, in the framework of a hierarchical exception handling scheme. Various UNIX-based components which have been implemented at Newcastle are used to illustrate this approach.

About the author

Professor Randell has been a professor of computing science at the Computing Laboratory of the University of Newcastle upon Tyne since 1969.

Suggested keywords

DISTRIBUTED SYSTEMS
NEWCASTLE CONNECTION
SYSTEM STRUCTURING
UNIX

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 681.322.06
SYSTEM DESIGN AND STRUCTURING FOR
DEPENDABILITY

B. Randell
Computing Laboratory,
University of Newcastle upon Tyne

ABSTRACT

The complexity of a large and sophisticated computing system can be
reduced significantly by ensuring that the system is constructed out of a
well-chosen set of largely independent components, whose normal and
exceptional interactions are properly defined and well-understood. However
the task of structuring a system, i.e. of choosing and defining appropriate
components, can be very difficult. This paper describes a technique of system
structuring which involves distinguishing the functionality which a system
is intended to have from other desirable attributes, including such aspects of
dependability as reliability and security. Separate components are then used
to provide each of these attributes, in the framework of a hierarchical excep-
tion handling scheme. Various UNIX-based components which have been
implemented at Newcastle are used to illustrate this approach.

1. INTRODUCTION

Computing System Design Methodology, as a subject, first rose to prominence in the
late 1960s. This was a time of growing concern in the profession about cost and schedule
over-runs on large computing projects, and about the inadequate performance and depend-
ability of many of the resulting systems. It was also a time when terms like ‘structured pro-
gramming’ and ‘software engineering’ first started to be bandied about. The aim, then as
now, was to find better methods of designing and implementing sophisticated computing
systems and, especially, of producing highly dependable computing systems. In this regard,
the particular issues on which I wish to concentrate on are the problems of coping with
complexity in systems design.

An obvious point, but nevertheless one worth repeating, is that the principal tech-
nique for coping with complexity is that of ‘divide and conquer’ - of somehow dividing up
the overall problem so that a designer, or design team, does not have to understand and
produce solutions for all the complexity all at once, and so that the resulting design itself is
constructed out of a well-chosen set of largely independent components, interacting in well-
understood ways. The crucial term here is ‘well-chosen’ - the problem is how to identify
appropriate components.

This problem manifests itself at a low level in the difficulties that inexperienced pro-
grammers often have in constructing programs for even relatively simple tasks, as they
learn that there is more to producing a well-structured program than just avoiding ‘go to’
statements, or even applying the latest proof-directed programming techniques. More
significantly, the problem is at the heart of some of the major difficulties facing project
managers and system architects starting out on a major new computing project. In such
situations, inadequate structuring can cost projects dearly in time and effort, and lead to
immensely baroque and redundant system designs, which are extremely difficult to main-
tain or extend, and which often possess only marginally acceptable performance and
reliability. Regrettably, one gains the impression that many current large systems are very poorly structured, and that much of their complexity and indeed size and unreliability is, so to speak, self-inflicted.

To describe the problem another way, structuring a system involves choosing the interfaces that are to be used within it. In some cases this can simply involve choosing already defined interfaces, supported by existing components. In complex systems it usually involves inventing new interfaces, i.e. creating new abstractions. This is far harder, at least to do well, since it is essentially similar to the tasks involved in inventing a new programming language.

Unfortunately, many, perhaps most, currently-active research areas in computing science and software engineering are, at best, peripheral to the topic of choosing a good system structure. For example, formal specification techniques, though of great value in documenting the intended functionality of each of the chosen components, as well as of the system as a whole, do not help with the actual choice of components. Some of the latest programming languages and computer architectures provide improved means of representing structure, but little help in choosing appropriate structurings.

Some software design methodologies do provide guidance with system structuring - for example Michael Jackson's scheme of requiring that the structure of file processing software be based closely on the syntactic structure of the input and output files[5], and its generalisation[6], which keys a system's structure to that of its environment. Another example, involving even more explicit concern with system structuring, is found in the 'Composite Design' technique[9] which is based on (subjective) assessments of the cohesiveness of individual components, and of the extent to which they are independent of each other. However what we would ideally like is some direct, and perhaps even automatic, means of devising, and providing objective assessments of the merits of, well-structured solutions to really complex computing system design problems.

Some years ago Peter Naur and I were each attracted to work, completely outside the computing field, which seemed to hold out promise in this regard. This was the scheme, proposed by an architect and town-planner, Christopher Alexander[1], for organising the task of planning a complete new town. His aim was to structure the design task, though not necessarily the end design, by automating the job of identifying the various design issues that should be tackled together, and of choosing the sequence in which these groups of design issues should be addressed. However it became clear on reflection that Alexander's scheme required, as input to his structuring algorithm, data that could only be provided by someone who had already successfully tackled many closely similar design problems, and who consequently would probably have little need of help with the structuring task. In fact, my search outside the computer field for sources of helpful ideas on design structuring ended rather ignominiously when I found an excellent book surveying methodologies of industrial design[7], one of whose principal conclusions was that industrial designers should now look to the field of computing science for methodologies which would help with the problems of tackling complex designs.

In fact even the assessment of the quality of the structuring of a given completed design is by no means straightforward. One can for example measure (or attempt to predict) the degree of coupling between components. (With hardware this could, for example, involve counting the number of wires between components and the amount of traffic they will carry; with software, it might for example involve assessing the frequency with which procedure calls are made, and the numbers of parameters involved.) Such a measure would presumably relate reasonably well to system performance, since it will reflect the overheads due directly to the structuring. However the quality of any structuring is also related to such apparently inherently subjective issues as (i) the complexity of the overall system, as compared to other systems of comparable functionality, and (ii) the likelihood that the various components will still be readily usable in differing circumstances and even in different systems. It is thus not at all surprising that fierce debates can arise over the
respective merits of rival proposed structurings of a given system, often through the unconscious use of differing criteria.

For all these reasons, it seems clear to me that the crucially important task of choosing an appropriate structuring for a computing system of any significant novelty and complexity will continue for some time to come to demand considerable human skill and experience - I referred earlier to the similarity between this task and that of inventing a programming language; there have been very few good yet novel programming languages invented in recent years.

There seems to be little hope of immediate and effective solutions to the general problem of producing a good overall system structure from a complex functional specification. However, certain design issues are amenable to reasonably objective general structuring strategies, which I would now like to discuss in some detail. In so doing, I will be drawing heavily on the work that I and my colleagues at Newcastle have undertaken in recent years within our project on 'The Reliability and Integrity of Distributed Computing Systems'. This project is a continuation of a long-term programme of research into system reliability, which has been funded over the years by the Science and Engineering Research Council and the Royal Signals and Radar Establishment.

2. DISTRIBUTED SYSTEMS

From the start our work has concentrated on the development of new structuring techniques, one of our first being the 'recovery block scheme'. This is a method of structuring a program so as to incorporate additional code (for purposes of providing a degree of tolerance to residual design faults) without adding unnecessarily to the program's complexity. The basic recovery block scheme was applicable just to isolated sequential programs. Over the years, whilst continuing to concentrate on fault tolerance as a means of achieving high system reliability, we have gradually extended the scope of our researches to deal with increasingly complicated systems, culminating in so-called 'distributed computing systems'. Such systems contain multiple computers, capable of acting, and failing, independently of each other, but also at risk from each other because of their interactions. (Such systems might, at one extreme, be constructed from mainframe computers linked by wide area networks, or at the other extreme, might coexist on the surface of a single VLSI chip.)

In fact I regret having to use the term 'distributed' computing systems, because I regard such systems as being the important general case. In contrast, centralised computing systems, and in particular systems which are either sequential, or which employ parallelism only internally, between interactions with their environment, are really rather limited special cases. Indeed, in industry and commerce the important problems of system design usually do not stop at the boundaries of a single computer system. Rather they also normally involve the activities of people and artifacts which interact with and through one or more computing systems, the whole forming a complex distributed system.

The first structuring technique which I want to discuss is a recent outcome of our continuing work on distributed systems. It is based on the idea that one should try to distinguish the functionality that a system is required to have, i.e. the intended relationships between its inputs and outputs, from other desirable attributes. One should then use separate components to provide each of these attributes. Examples of non-functional attributes, or 'abilities' as we have come to call them, are distributedness, performance, reliability and security. To the extent that abilities are truly independent of the system functionality, so the corresponding components can be 'transparent', i.e. can be inserted into or removed from the system without affecting other components. Of these abilities, distributedness is, in some sense the most basic, as I will try to demonstrate.

For fear that this all sounds rather vague and impractical, I will describe, albeit briefly, some of the systems we have actually implemented with the aid of this structuring technique. All our systems have been based on UNIX† - for a particular reason. This is

† UNIX is a trademark of AT&T Bell Laboratories.
that UNIX though far from perfect has, we have discovered, a rather special property: although it was designed as a multiprogramming system, running on a single computer, its functionality is equally appropriate for a distributed system. The principal characteristics of UNIX that make this the case are:

(i) the fact that it provides users and their programs with multiple processes, which can appear to operate in parallel, and

(ii) its naming facilities (i.e. the means it provides for identifying its various constituent objects, such as devices, files and programs) are strictly contextual, and therefore independent of whether a UNIX system is a complete system, or merely a component of some larger system.

Such strictly contextual naming is not common in the world of computing systems, despite the fact that it is well-known elsewhere, particularly in telephone systems. Indeed such systems possess a number of characteristics that the designers of distributed computing systems would do well to copy. In telephone systems, telephone numbers act as a set of names. Their hierarchical organisation ensures that names are contextual. Thus the telephone numbers used in a company's internal telephone system, for example, need not be affected if the system becomes part of a national telephone system. National telephone numbers need not be changed if the country becomes part of the international telephone system, etc.

In computing system terms, a full 'contextual naming scheme' is one in which all names are context-relative, and which has means for introducing new contexts, and for entering and leaving naming contexts. This is a characteristic that UNIX possesses by virtue of its very simple yet general scheme for naming files, devices and programs, in which directories serve as the required contexts.

```
  root
   /\  \
  /   \ /
user lib
   /\  \
  /   \ /
 current working directory
   /\  \\
  /   \ \
       /\  \\
     /   \  \\
    /     \  \\
   /       \  \\
  /         \  \\
/           \  \\
dir1
   /\  \\
  /   \  \\
 a    b
```

Figure 1: A Typical UNIX Name Space

Figure 1 shows part of a typical UNIX naming hierarchy. Files, directories, etc., can only be named relative to some implied "location" in the tree. It so happens that UNIX provides two such locations, namely the directory which is designated as being the "current working directory" and that which is designated as the "root directory". Thus in the figure "/user/brian/dir1/a" and "dir1/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 that "." indicates the 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 relative to the current working directory rather than the root directory.

The root directory is normally positioned at the base of the tree, as shown in the figure, but this does not have to be the case. Rather, like the current working directory, it can also be re-positioned at some other node in the naming tree, but this position must be specified by a context-relative name. Thus all naming is completely context-relative - there is no means of specifying an absolute name, relative to the base of the tree, say. (The base 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 hierarchical naming scheme. It is for these reasons that UNIX, in contrast to most other operating systems, can be said to support a contextual naming scheme.

We have developed means for linking together a number of computers, each running a conventional UNIX system, to form a distributed system which is functionally indistinguishable from UNIX, and which we have termed a 'UNIX United' system. The UNIX 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.

Figure 2 shows 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. (This naming structure need bear no relationship to the actual topology of the underlying communications networks. Indeed this exact naming structure could be set up on a single conventional UNIX 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".

![Figure 2: A University-Wide UNIX United System](image)

UNIX United systems can themselves be used, in exactly the same way, as components of a yet larger UNIX United system. We are therefore using UNIX as the basis of a recursive method of system construction. We thus gain the usual benefits of recursive structure, namely conceptual simplicity and extensibility. For example, in Figure 2, U2 and the directory structure beneath it might not be associated with a single machine. Rather it might be a UNIX United system, itself containing an arbitrary number of other
UNIX United systems, unknown to U1 in CS. (One can draw an analogy to the way in which ALGOL 60, and its successors, allow statements to be grouped together to form a single statement. In contrast its predecessors, FORTRAN and COBOL, had only very limited means of building up program texts.)

Construction of a UNIX United system involves linking the various machines together physically, via one or more networks, and installing a new software component in each system. This component, which we could not resist calling 'The Newcastle Connection', is transparent, both in the sense that users and their programs do not need to be aware of its presence, and in that it can be installed without having to make any changes to the component UNIX systems.

The positioning of the Connection 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. These are 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 therefore, in essence, inserted between the kernel and the processes (see Figure 3). From above the Connection layer 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, as remote procedure calls, to another UNIX system (for example, because they concern files or devices on that system). Similarly, it accepts calls that have been re-directed to it from other systems.

![Diagram](image)

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

Although this technique of constructing, and structuring the software of, a distributed computing system has been described very much in terms of UNIX, it is potentially more generally applicable. Indeed, we believe that the concept of a recursively structured distributed system can be applied at other levels of system design - currently another research group at Newcastle is investigating it in connection with processor architectures and the geometrical structuring of VLSI designs. However I wish now to leave the issue of 'distributedness', and go on to discuss the provision of other non-functional system attributes, or 'abilities' as I have termed them.

### 3. PERFORMANCE AND RELIABILITY

First a few words on performance, again for convenience couched in terms of UNIX United. The Newcastle Connection does not contain any sophisticated algorithms for processor allocation and scheduling, e.g. to perform load-balancing. Rather it simply arranges that each process is run on the processor associated with the file store in which its code is
held. However system administrators can re-organise a UNIX United system so as to incorporate additional computers, without changing the appearance of its overall naming structure, and hence without requiring users to change their programs. By this means the performance of the overall system, and of many individual programs, can be augmented simply by turning what had previously been quasi-parallel processing into actual parallel processing. Moreover, given the flexibility afforded by the Newcastle Connection approach, it would be comparatively easy to add, without changing any existing software, mechanisms for load balancing which in suitable circumstances might provide considerable additional performance improvements.

However my colleagues and I have been paying more attention to another 'ability', namely 'reliability'. One of several reliability mechanisms which we have investigated is the use of 'Triple Modular Redundancy' at the level of complete UNIX systems, as a means of masking hardware crashes and malfunctions. 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 Triple Modular Redundancy layer) to each of a number of UNIX machines on top of their Connection layers, as shown in Figure 4. The TMR layer goes on top of the Connection layer because it can then rely on the latter to handle all problems relating to the physical distribution of processes, files, etc. Copies of a conventional application program and its files can then be loaded onto each of three machines and run so that file accesses are synchronised 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. The idea of majority voting and re-configuration is of course not a new one. The point is that the technique is very simple to implement when it is separated from issues of distributedness.

Figure 4: Hardware Fault Masking

The use of such 'masking redundancy' is in fact but one step towards a more general approach to the construction of fault tolerant systems, which uses a second structuring technique. This approach is more sophisticated in that it does not assume that all faults are successfully masked at some particular level. Instead it allows for defences in depth, via the use of hierarchical error detection and recovery mechanisms to supplement any attempted fault masking. The structuring technique allows, in principle, each system component at each level to signal exceptions in order to report, to the next higher level, any fault which it cannot, or is not designed to, mask and to receive similar exceptions reported to it from lower levels. In general, each such component can contain means of attempting
(not necessarily completely) to handle (i) faults reported to it by sub-components, (ii) faults within its own design, and (iii) faulty requests for services from its environment. In practice, engineering judgement will be used to determine just what fault tolerance should be incorporated in which components, and what exceptions should be defined and handled. (For a fuller treatment of this approach to the structuring of fault tolerant systems see[2], from which Figure 5, illustrating the inter-component interactions implied by the approach, is taken.)

![Diagram](image)

Figure 5: A generalised Fault Tolerant Component

Using such an approach, one might, for example, provide an additional error recovery layer based, say, on the use of checkpointing, in each of the UNIX machines of Figure 4, which could be invoked when no majority consensus can be formed, presuming that reconfiguration is not provided, or does not succeed. Moreover, it would be appropriate to design such an error recovery layer so that it could also be used in an isolated machine, when no TMR layer was present.

4. SECURITY

A third characteristic that can, to a great extent, be considered as being independent of the functionality of a system is 'security', i.e. the enforcement of rules about information usage. In particular, military environments often require the enforcement of so-called multi-level security policies, governing the storage and transmission of information that has been classified into, for example, 'top secret', 'secret' and 'restricted' grades.

In such environments it is regarded as essential to have a system which has been constructed so 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 have been proven formally. Particularly important in this regard are the mechanisms which prevent illegal information flow, and the mechanisms which monitor and mediate all allowable information flow between system components which for one reason or another cannot be trusted to adhere to the security policy. (For example, they might still contain residual bugs, or in fact have been supplied by Trojan Horse Software, Inc.)
A recursively structured distributed computing system provides an ideal environment in which to use such security mechanisms - one can allocate (untrusted) general-purpose component computers to different security levels and implement appropriate (trusted) security mechanisms as transparent additions to the inter-processor communication links. Figure 6, which is taken from Rushby and Randell[11], portrays such a system. It shows a set of untrusted UNIX systems, each linked to a local area network via a 'Z-box', or 'Trusted Network Interface Unit', as it is known in the trade. These boxes employ encryption techniques to prevent information flow between security regimes and to control the types of security reclassification allowed. They are to a very large extent transparent to ordinary users, who therefore have the impression that they are using an ordinary UNIX United system, indeed an ordinary UNIX system. (Further details of this approach, and of a prototype version of such a system, constructed at RSRE Malvern based on UNIX United, are given in[12].)

Security has thus been introduced, as an additional ability, merely by the incorporation of additional components, which are (at least relative to UNIX itself) very simple, and hence amenable to formal specification and verification. I should stress though that the idea of using TNIUs to construct a multi-level secure distributed system is not new - what is significant here is that it is being allied with a means of handling distributedness which allows users to disregard the fact that the system is distributed.

However, once again, it is appropriate to point out that the above TNIU scheme can, like the TMR scheme, be generalised, and in essentially the same way. This is hardly surprising, once one accepts the view that reliability and security are just special cases of a more general concept, namely that which we and others have come to term dependability[8]. Dependability can be achieved by fault prevention and/or fault tolerance, and fault tolerant systems can in general be structured so as to use fault masking techniques (i.e. static redundancy) or error detection and recovery techniques (i.e dynamic redundancy).

Thus security can be regarded as the provision of certain forms of dependability (concerned with information usage, rather than functionality), particularly in the face of intentional faults caused by would-be system penetrators. Though virtually all work to date on (computer) security has concentrated on fault prevention, by testing or formal verification, fault tolerance techniques could also have a part to play, particularly given the similarity
of the problems caused by intentional faults and by residual design faults. (This argument is given in much more detail in[3].)

The actual generalisation of the TNIU scheme which I have in mind is one in which it is not assumed that the TNIUs are totally and solely responsible for all security provisions. Rather, some of their security responsibilities might be devolved to sub-components, and the TNIUs themselves might have means of reporting security faults which they cannot mask to some higher authority. All of this can be done without losing the ability to construct systems recursively, as long as the interface chosen as the basis of the recursion includes adequate means of exception reporting (as does in fact the UNIX system call interface).

Thus, since the two mechanisms I have outlined, for providing reliability and security as additional system 'abilities', are independent of, and compatible with, each other, one should therefore be able to produce, very easily, an apparently ordinary UNIX system which provides the twin abilities of high reliability and high security, without modifying either type of mechanism, or having to re-validate the security mechanism.

5. CONCLUDING REMARKS

What I and my colleagues have been attempting to do is to develop and evaluate the effectiveness of some particular types of system structuring. In line with Professor Edsger Dijkstra's strictures on the need to achieve "separation of logical concerns", we have investigated the notion of (i) separating the characteristics required of a system into its functionality, and various distinct abilities, and then (ii) using such separation as a direct guide to structuring the system. The principal system component that we have developed is one that provides the 'ability' that I have termed 'distributedness', i.e. that makes a set of hitherto independent systems function as a single coherent system.

The treatment of 'distributedness' as an ability, separate from a system's functionality, leads naturally to the achievement of what is often called 'network transparency'. Many groups have built distributed systems which exhibit network transparency, to a greater or lesser degree. What I have described differs from the work of most others in the stress that we have put on recursiveness. Right from the start we automatically sought a mechanism for joining a set of computer systems together to form a distributed system, which would be equally capable of joining distributed systems together. (I should explain that, no doubt as a result of my exposure to Algol 60 at a tender and impressionable age, I have long held that "in computing science, if an idea is any good, it will be even better if it is made recursive").

This is not just academic pedantry, but is, I claim, an intensely practical point. It provided us with an acid test of how completely we had achieved network transparency. It also provided valuable guidance concerning how various mechanisms within the Newcastle Connection should be designed. Moreover it has been central to the approach we have taken to the transparent provision of other system abilities, including several types of dependability. But perhaps above all it is what I would call a "simplifying generalisation" - and that alone is sufficient justification for recursiveness, as far as I am concerned.

The second point that I wish to make is that the computer science world has concentrated unduly on the design of sequential and centralized systems. This attitude is perhaps the true "von Neumann bottleneck" - a term which is usually applied just to the narrow interface between processor and memory which is characteristic of von Neumann computers, and which in certain research circles it is now so fashionable to criticize.

To my mind, von Neumann machines have served, and are likely to continue to serve us, well. What is needed are improved means of constructing systems using multiple computers of whatever type - systems which will possess, almost inherently, much greater flexibility and resilience than any single sequential computer can ever hope to achieve. After all, the environment of people, procedures, machinery, etc., that a typical large scale
The computing system is intended to fit into, and contribute to, is itself a distributed system, usually possessing an enviable degree of flexibility and resilience - at least until the computer arrives!

The problems of specifying, and implementing, asynchronous algorithms are obviously a central issue here, though one on which much good work has been done in recent years, particularly in the UK. On the other hand, naming and exception handling mechanisms are equally relevant, and have not received the amount of attention they deserve - all too often naming mechanisms, in particular, have been studied just from the viewpoint of a single computer, or a single program. In more general environments, for example many multiprocessing systems, and also most computer networking systems, the treatment of these issues is often extremely ad hoc. Even the naming techniques which I described earlier, though central to our work, are to be regarded as adequate, rather than ideal.

The notion of a recursively structured distributed system is really a very simple one and hence of possible wide applicability, as indeed is that of distinguishing functionality from other required characteristics, and of providing a general exception handling hierarchy, so as to structure systems accordingly. We do have clear evidence that these ideas work well at the operating system level, using computers connected by local area networks, where the modest overheads introduced by the structuring seem well worthwhile. The fact that most existing operating systems do not possess the required naming and asynchrony characteristics is obviously unfortunate, to say the least, though in many cases one could at least incorporate computers running such operating systems into an overall recursively distributed computing system, as somewhat second-class citizens, so to speak. Much more interesting is the possibility of applying these structuring techniques to major application-level programs, such as large real time systems. All these topics, I would claim, both require and merit further research.

6. ACKNOWLEDGEMENTS

This paper is based closely on an earlier paper[10], but also incorporates some material from a paper co-authored by John Dobson[3]. The preparation of both these papers has been greatly aided by discussions with colleagues at Newcastle, at RSRE, and in IFIP Working Group 10.4.

References


