System Design and Structuring

B. RANDELL

Computing Laboratory, University of Newcastle upon Tyne, Claremont Road, Newcastle upon Tyne, NE1 7RU

The task of implementing a large and sophisticated computing system is often unduly costly and time-consuming, with the resulting system exhibiting inadequate performance and reliability, because of excessive system complexity. Such complexity can be reduced significantly by ensuring that the system is constructed out of a well-chosen set of largely independent components, which interact in well-understood ways. 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, such as reliability and security, and then using separate components to provide each of these attributes. Various UNIX-based systems which have been implemented at Newcastle are used to illustrate this structuring technique.

Received March 1985

1. INTRODUCTION

At the time the British Computer Society was formed I was earning my living by programming a computer whose order code was all too obviously designed largely by Alan Turing. And I was programming it using machine language, and binary machine language at that, not even assembly language. Yet this computer, whose use I shared with many other people, is greatly surpassed, with respect to memory capacity, computational power and programming convenience, by the one that I bought as a Christmas present for my children just over a year ago, and carried home in a plastic shopping bag!

However, the subject of system design and structuring is one which has made rather less spectacular progress. 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 overruns on large computing projects, and about the inadequate performance and reliability of many of the resulting systems. It was also a time when terms like 'structured programming' 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. In this regard, the particular issues on which I wish to concentrate here are the problems of coping with complexity in systems design.

An obvious point, but nevertheless one worth repeating, is that the principal technique 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 in the difficulties which inexperienced programmers 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 contorted system designs, which are extremely difficult to maintain or extend, and which often possess only marginally acceptable performance and reliability. Regrettably, one gains the impression that many of today's large systems are very poorly structured, and that much of their complexity and indeed size is, so to speak, self-inflicted.

To describe the problem another way, structuring a system involves choosing the interfaces which are to be used within it. In some cases this can simply involve selecting 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. Similarly, some of the latest programming languages and computer architectures provide improved means of representing structure, but little help in devising appropriate structurings.

Some software design methodologies do attempt to provide direct 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, 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, 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 one 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 some 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, for organizing the task of planning a complete new town. His aim was to structure the design task, though not necessarily the end design, by automatizing 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 which could only be provided by someone who had already successfully tackled many closely similar design problems. Needless to say, such a designer 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 somewhat ignominiously. This was when I found an excellent book surveying methodologies of industrial design, whose principal conclusion was that industrial designers should now look to the field of computing science for methodologies which would help with the task of tackling complex problems.

In fact the assessment of the quality of the structuring of a given completed design is by no means straightforward. One can for example actually measure (or attempt to predict) the degree of coupling between components. 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 different assessment criteria.

For all these reasons, it seems clear to me that the crucially important task of choosing an appropriate structure 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 very yet novel programming languages invented over recent years.

There seems little hope therefore 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 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 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 co-exist 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 (i.e. in 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 boundary 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 structuring technique which I wish 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 which a system is required to have, i.e. the intended relationships between its inputs and its 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 briefly describe 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 that Unix though far from perfect has, we have discovered, a rather special property: although it was designed as a multiprogramming

---

* Unix is a trademark of Bell Laboratories.
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, one of my favourite examples being the telephone system. Indeed such systems possess a number of characteristics which 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 which 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 --> brian fred |
  /\        |
 directory |
  /\        |
 dirl b c  |
  /\        |
 a b       |
```

Figure 1. A typical Unix name space.

Fig. 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/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 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 actual base of the tree, say. (The base directory can only be recognised by the convention that it is its own parent.) Moreover all other means provided for identifying any of the various kinds of objects which Unix deals with, e.g. users, processes, open files, etc., can be 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.

Fig. 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 which the Electrical Engineering Department has also chosen to call ‘U2’ will need to be identified with names starting ‘/…/EE/U2’.

```
CS EE Maths...
  /\ /\ /\ 
 U1 U2 U1 U2 U3
```

Figure 2. A university-wide Unix United system.
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 Fig. 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 – incidentally celebrating its actual Silver Jubilee this year – allows 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 Fig. 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 which have to be redirected, as remote procedure calls, to another UNIX system (for example, because they concern files or devices on that system). Similarly, it accepts calls which have been redirected 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 recursiveness 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 Connection layer does not contain any sophisticated algorithm 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 reorganise 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. In suitable circumstances these could 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 component (the Triple Modular Redundancy layer) to each of a number of UNIX machines on top of their Connection layers, as shown in Fig. 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 on to each of three machines and run so that file access 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 reconfiguration – or should I call it reselection – 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.

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 which has been classified into, for example, 'top secret', 'secret' and 'restricted' grades.

---

**THE COMPUTER JOURNAL, VOL. 29, NO. 4, 1986 303**
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 proved 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 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. A prototype of such a system has in fact already been constructed at RSRE Malvern, based on UNIX United, following earlier experiments at Newcastle. This system is portrayed in Fig. 5 below. This 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 often known. 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. (Such details as are available about this work are given in Rushby and Randell.)

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 Z-boxes 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, perhaps the most important point to be made is that 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

I will now attempt to summarize the main points which I have been trying to make, and to reiterate (perhaps belatedly) my belief that the techniques which I have been describing in terms of a particular fashionably controversial) operating system are potentially of much more general applicability.

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 those of various distinct abilities, and then (ii) using such separation as a direct guide to structuring the system. The principal system component which we have developed is one which 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 which we have put on recursiveness. Right from the start we sought a mechanism for joining a set of computer systems together to form a distributed system, which would be equally capable of joining such distributed systems together. (I should explain that, no doubt as a result of my early exposure to Algol 60, I have long held that "in computing science, if an idea is any good, it will be even better if 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 Connection layer should be designed. Moreover, it has been central to the approach we have taken to the provision of other system abilities, such as reliability and security. But perhaps above all it is what I would call a "simplifying generalisation" and that alone is sufficient justification or 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 centralised systems. This concentration 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 criticise.

To my mind, von Neumann machines have served us well, and will continue to do so. 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 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, parallel programs 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 mechanisms are equally relevant, and have not received the amount of attention they deserve - all too often they have been studied only from the viewpoint of a single computer, or a single program. In more general environments, for example multiprocessing computers and computer networking systems, the treatment of naming issues is often extremely ad hoc. However, even the naming techniques which I described earlier, though central to our work, are to be regarded as adequate, rather than ideal. This topic is thus one which certainly requires further research.

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 structuring systems accordingly. We do have clear evidence that these ideas can 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 parallel processing characteristics is obviously unfortunate, to say the least, though in many cases one could at least incorporate such operating systems into an overall recursively distributed system, as somewhat second-class citizens, so to speak.

Much more interesting, though, is the possibility of applying these structuring techniques to application-level programs, such as large real-time systems. After all, as I explained earlier, the principal motivating force behind our work is a concern for the great, and often unwarranted, dependence that people and governments are tending to place on various immensely large and sophisticated computing systems. However, at this stage, all that I would presume to claim is that such an application merits additional research.

REFERENCES


