Reliability and security issues in distributed computing systems.

B. Randell and J. E. Dobson

Abstract
We analyse a number of interesting actual or potential parallels between the problems and techniques associated with achieving high reliability, and those associated with the provision of security, in distributed computing systems.

(Invited paper, Fifth Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, C.A. January 1986.)
Bibliographical details

RANDELL, Brian

Reliability and security issues in distributed computing systems. By B. Randell and J.E. Dobson.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 210.)

Added entries
DOBSON, John Edward.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract
We analyse a number of interesting actual or potential parallels between the problems and techniques associated with achieving high reliability, and those associated with the provision of security, in distributed computing systems.

About the author
Professor B. Randell has been a Professor of Computing Science in the Computing Laboratory since 1969.

Mr. J.E. Dobson was a systems programmer in the Computing Laboratory from 1974 until June 1980. He is now Technical Director at MARI, Newcastle upon Tyne.

Suggested keywords
DISTRIBUTED SYSTEMS
FAULT TOLERANCE
RELIABILITY
SECURITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 658.47
U. D. C. 519.687 519.718
RELIABILITY AND SECURITY ISSUES IN DISTRIBUTED COMPUTING SYSTEMS

B. Randell
Computing Laboratory,
University of Newcastle upon Tyne

J. E. Dobson
Microelectronics Applications Research Institute,
Newcastle upon Tyne

ABSTRACT

We analyze a number of interesting actual or potential parallels between the problems and techniques associated with achieving high reliability, and those associated with the provision of security, in distributed computing systems.

(Invited paper, Fifth Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, CA, January 1986.)

1. INTRODUCTION

The field of computing has grown so rapidly, and to such an extent, that it has given rise to a large number of different sub-cultures, each with its own jargon, its own research symposia, and its own (often blinkered) view of the overall field. For example, under the banner of "reliability" tend to gather people who are concerned with the continuous functioning of aircraft flight control systems, the correct operating of networks of cash dispensers, etc. A largely separate group of researchers and system designers march to the "security" drum, mainly at the behest of the military, and other more reticent government agencies. Such divisions are understandable - but this does not make them totally beneficial.

In this paper we propose to examine what we believe to be interesting, and potentially important, links between these two apparently distinct topics, and indeed will argue, as has been implied by Laprie[1], that they can usefully be regarded merely as different aspects of a common problem, and so susceptible, at least in part, to common solutions. In so doing we will be taking an apparently embarrassingly different line to that taken when the first author was, once before, an invited speaker at this symposium. At Clearwater Beach he argued[2], we would like to think eloquently, for an approach to system design which involved treating various issues, including reliability and security, as logically separate problems, provisions for which should be made by logically separate mechanisms. Indeed a number of actual UNIX-based experimental
systems based on such an approach were described. In fact, however, the inconsistency between these two papers is, we hope to show, more apparent than real.

Some links between the subject of reliability and that of security are of course obvious, and well-acknowledged. Most notably, one would, at very least, expect that due effort is spent ensuring adequate reliability of the security-critical mechanisms in any system which purported to meet any significant security criteria. Also, one would surely normally expect information which is extremely secret also to be extremely valuable, and so require it to be held reliably, as well as securely. Quite how this is done without the reliability mechanisms compromising the security mechanisms is another matter! Indeed, it was such concerns that prompted the stress on separability of security and reliability mechanisms in the earlier paper.

The mechanisms discussed in that paper included an NMR-type reliability mechanism and an encryption-based security mechanism, added to each of a set of computers in a distributed computing system. The particular implementation described was based on UNIX† and made use of software developed for constructing so-called UNIX United distributed systems, i.e. the Newcastle Connection[3, 4]. The unusual degree of independence between the reliability and security mechanisms achieved was greatly facilitated by the recursively structured nature of UNIX United – its functionality is identical to that of each of its component UNIX systems. In contrast, like the Newcastle Connection, the reliability and the security mechanisms have no visible functionality; they improve the quality, without affecting the nature, of the service that the component UNIX systems provide. One can thus remove the reliability mechanisms without affecting the security mechanisms, and vice versa – a strong indication that they are indeed completely independent of each other. Nevertheless, as we have indicated, we now feel that, as concepts, reliability and security are not necessarily best treated so separately, and that their joint consideration can lead to some interesting new insights.

2. DISTRIBUTED SYSTEMS

Before launching into the discussion of reliability and security, we must first explain why we have linked our paper specifically to "distributed" computing systems. First, for the record, by this term we mean a system made up of multiple computers, interacting via a non-instantaneous communications medium, capable of working (and failing) independently of each other. Such systems might be spread over a large geographical area, or, in the not-too-distant future, might coexist on a single silicon wafer or even chip – in other words, the spectrum stretches from the world of OSI to that of VLSI!

In fact our real interest is in "distributed systems", i.e. systems which could, for example, include people and machines that are interacting with, and perhaps through, one or more computing subsystems. Against such realities the problems of an isolated sequential computing system look very specialised and uninteresting, and even treatment of distributed computing subsystems, without considering the human environment in which they are placed, can sometimes induce an enervating feeling of unreality.

† UNIX is a trademark of AT&T Bell Laboratories.
In practice, since we have a recursive view of systems as being constructed from components which themselves will often usefully be regarded as systems, the question of whether a given system is distributed is level-dependent. Many an apparently conventional sequential computer actually contains, nowadays, multiple interacting microcomputers. Similarly, it can well be being used to support, apparently simultaneously, a set of processes whose means of interaction are sufficiently constrained for the set to be usefully viewable as a distributed system.

The relevance of distributed computing systems to reliability is well-known. Though they pose additional reliability problems, they provide the basis for a variety of reliability mechanisms. The essence of these mechanisms is redundancy and separation - the provision of additional separate means of storing, communicating and/or processing information, so that errors can be detected and recovered from, or masked, so that failures can be averted. However as Rushby[5] and others have argued, "separation" is also at the heart of the security problem - and manifest physical separation is one of the simplest and most easily verifiable forms of separation.

It is possible to try and build complex software and hardware mechanisms which will guarantee that a given single computer system prevents, say, top secret information leaking into unclassified files. However one can, we believe, more easily achieve an equivalent effect by assigning the separate component computer systems of a distributed computing system to separate security regimes, in the knowledge that the only security-critical mechanisms will then be those relatively simple ones that are involved with inter-computer communication.

This is what has been done, at the Royal Signals and Radar Establishment in the U.K., where a prototype distributed multi-level secure system has been constructed, using UNIX United, and based on the design outlined in [6]. Oversimplifying somewhat, each separate UNIX system looks like a directory in the overall system, and runs at a single security level. The rules concerning permissible information flows between levels are enforced by placing the trusted mechanisms for enforcing security policy in specially designed network interface units, which are also responsible for appropriately encrypting all inter-machine information communication. As a result, no trust has to be placed in any of the individual UNIX systems[7]. Moreover, the overall multi-level secure system still appears to its users to be a conventional UNIX system.

3. RELIABILITY

During recent years, the Newcastle and other research groups have spent much time seeking improved definitions for the various fundamental reliability concepts. Our own dissatisfaction with the hitherto typical definitions of terms such as "failure", "error" and "fault" arose early on, when we started to consider the possibility of providing what is now termed "design fault tolerance", particularly for software. We found it inappropriate to start from an enumerated list of possible faults, which had been the hardware engineers' approach - this makes little sense when one wishes to allow for the possibility that design faults, of unknown form, are still lurking somewhere in the deeper recesses of the system. Instead we took the notion of a "failure", i.e. the event of a system deviating from its specified behaviour, as our starting point, and then defined "reliability", "fault" and "error" in terms of "failure"[8].
With such an approach, if one has more than one specification for a single system, each capturing some different requirement which is to be placed on its operation, one has thereby defined different types of failure, and indeed different types of reliability. However, to avoid unnecessary confusion, we will follow the lead given by Laprie and use the term "dependability", to encompass the different types of "reliability". This will enable us to adhere to the more common informal usage of reliability as relating to functionality and the continuity of its achievement.

One can, for example, have a specification which concerns the "safe" behaviour of a system, deviation from which specification could cause failures likely to endanger human life. Thus "safety" is merely a slightly different special case of "dependability", for which one can use all the reliability definitions, and many of the design arguments, virtually unchanged. Indeed we have at times been tempted to produce papers for conferences on "safety" merely by using a simple context-editor on previous Newcastle papers on reliability problems and mechanisms - a temptation which, we hasten to add, has been resisted!

Security requirements, such as rules for enforcing multi-level security, similarly result in a (partial) specification of the behaviour of a system, so that in these terms one can again regard security as a special case of dependability. This is in essence the viewpoint we will adopt in this paper. All this may seem like playing with words. However we have found the viewpoint a very helpful one, in that it has led us to an initial, yet interesting, exploration of some of the parallels (and divergences) between the reliability and security areas. We will develop this point further, in the following section, before attempting to discuss what we believe to be the more fundamental issues that lie behind the reliability/security analogy.

4. SECURITY

The principal techniques for trying to achieve reliability can be classified into fault prevention and fault tolerance - often regarded as rival, but more profitably as complementary, techniques. Fault prevention attempts to ensure that an operational system is fault-free, either because any faults it contained were removed before putting it into service, or by avoiding the inclusion of faults from the outset. It is very noticeable to us that work on secure computing systems has typically concentrated on the use of (security) fault prevention techniques. The use of penetration exercises, for example, can be the basis of a fault removal strategy. However the preferred approach is to try to avoid faults, by insisting on formal or semi-formal analysis of the specification at each level, and verification that it is met by the design and its implementation. But this concentration on fault prevention, and in particular on fault avoidance, seems to apply only to the computer component of a secure environment. In a wider environment, one would expect to find plans and protocols for recognising and dealing with security leakages. After all, capturing and turning spies, for example, is in effect a form of fault location and error compensation, i.e. of fault tolerance in practice. It is perhaps surprising that more attention has not been placed on such techniques in the design of secure computing systems.

However, before we go on to discuss the possible applicability of fault tolerance techniques to the problem of ensuring the security of a computing system, it is appropriate to consider the role of a secure computing system as a component within a secure enterprise, alongside the other components such as sentries, hidden cameras,
barbed wire, or whatever. One expression of the requirement for its introduction might be that the computing system introduces no new security flaws into the overall system, i.e. that it is in itself inherently secure. There are two criticisms that can be made of such a requirement. The first is that it is unrealistic to believe that it is possible, just as it would be unrealistic to believe that the introduction of a new and complex component into a system would not alter the reliability characteristics or would not introduce any new failure modes into the system; and the second criticism is the system structuring one that the whole system should have a recursive reliability or security failure model with well-defined exception interfaces for the signalling of faults/failures and that all components should conform to this model.

When these criticisms are taken into account, it can be seen that one task of the secure system designer is to consider what can be done entirely within the computing system, and what aids can be given to the people in the environment of the system to assist in the forward and/or backward error recovery procedures necessary to limit damage following a security violation. (The problem is similar to that of assisting a database system user who finds out that his database has for some time contained, and has been giving him, incorrect data[9].)

In summary, what we are saying is that one should attempt to employ a methodology based on building secure systems out of insecure components, or more accurately, less insecure systems out of more insecure components. Each level of component, and the overall system, would have its own security specification. This will characterise the desirable behaviour of the component at its interfaces with its environment, and also define any means it is designed to have of signalling exceptions. In principle any such specification should aim to be as complete as possible — in practice one might choose to abbreviate the specification by making it explicitly dependent on the security specifications of the sub-components (and by assuming that these are not violated)[10].

Whenever it is admitted that inadequately secure components are being used, then it will be appropriate to try and design means of masking, or of detecting and recovering from, the security errors which might arise. When the fear is that the design (whether or not it is responsible for dealing with sub-component insecurity) might itself be inadequately secure, despite whatever efforts have been made at formal verification, then attempts might be needed to make the design itself fault tolerant. This is in fact not as novel a proposal as one might at first think — security mechanisms are normally expected to defeat deliberate attack, unlike most (conventional) fault tolerance schemes. However an important characteristic of a deliberate attack is that the point of failure cannot be usefully predicted, even on a probabilistic basis. This is however also a characteristic of residual design faults, so that schemes which provide a measure of design fault tolerance, such as N-version programming and recovery blocks, as well as Byzantine agreement protocols[11], are perhaps all of potential relevance to the problem of defeating deliberate attacks. In practice though it can be admitted that the security problem is the more severe, since the people mounting attacks on a would-be secure system may well find means of making use of their knowledge of the very mechanisms that are intended to defeat them.

In support of this view that security mechanisms can be seen as, or perhaps even derived from, fault tolerance mechanisms, let us cite a number of additional analogies:
(i) A common technique for the structuring of software to be used in reliable systems is to analyse the functionality in terms of atomic actions which either succeed or fail completely, thus permitting clean well-structured recovery mechanisms. One perhaps could use the same or analogous methods for analysing the functionality of secure systems in terms of security regions across whose boundaries guarantees can be given with respect to information flow.

(ii) A system could be designed with internal security checks (e.g. based on the use of capability mechanisms) which will enable at least some types of security violation to be prevented, and the attempt reported on, just as a fault tolerant system might contain internal error detection and exception signalling mechanisms. Such internal security checks could enable one to rely on less complete formal security proofs, by analogy with the notion of “safe” programming[12].

(iii) One could perhaps use a direct analogy to the concept of a generalised fault-tolerant component, and the corresponding three-way classification of faults with respect to a given component[13]. Thus a given component would return an exception if asked to do something outside its remit which would compromise security, and it would attempt to mask its own security errors, and to handle, and if possible mask, any security exceptions signalled to it by components it itself is using.

(iv) A system’s outputs could be, at least partly, vetted for security violations (e.g. by a real or an automated security watch officer) just as they might be subjected to acceptance tests for the purpose of determining whether to invoke error recovery.

(v) An important task for a security watch officer is to confirm that the system has not performed completely unintended actions. This is in essence the frame problem[14]. Some recovery block implementations have provided assistance with this problem by requiring that acceptance tests must not only evaluate to true but must also access all the variables that have had assignments made to them, in order to succeed.

(vi) Dependable constraints on, or mechanisms for the recording of, information flow can provide a basis for dealing with the situation when it has been determined that one or more security violations have occurred. They can be used to determine what purging mechanisms should be invoked (i.e backward error recovery) or, at least in principle, what misinformation should be transmitted where (i.e. forward error recovery by compensation).

(vii) Encryption is in essence just a means of protecting information in a hazardous environment, and as such can be viewed as a reliability mechanism.

We have looked at the analogy between reliability and security and indicated ways in which reliability thinking should influence the designers of secure systems. We now turn the question around and ask: Have the designers of reliable systems anything to learn from security thinking?

The answer seems to be: Not a lot! This is partly for the accidental reason that security thinking is more recent and therefore less developed, and partly for the more general reason that considering security as a particular case of dependability, it will inherit from above more than it can return from below. There are, however, two or three
security concepts which would seem at first sight to have application in the reliability area.

The first is the notion of an unforgeable number as a way of detecting certain kinds of fault. (A set of unforgeable numbers is a set which is very sparsely distributed over a large space, so that it is easily checked whether a particular number is a member of the set or not, but a number chosen or created at random is almost certain not to be.) It would be possible to create a machine instruction set whose only valid instructions were unforgeable numbers, which machine would then have the property that overwritten or otherwise corrupted programs would be trapped immediately instead of being executed (correctly, as far as the hardware is concerned) until some other error was detected or failure occurred. However it would probably be better, and would certainly be more economical of storage, to use a compactly coded instruction set, together with explicit defences against program corruption.

The second and third applications derive from the notion of authentication. In terms of a simple client–server relation, the client has to authenticate himself to the server as being allowed to use the service provided, and may well have to prove that he is the same client as requested some previous service. One common way is for the client to be issued with a token which is used to prove his authenticity. The form of the token may vary, but its main property is that the server can quickly recognise whether or not it is valid.

For example, it may be important to provide a particular server with stable memory for purposes of crash resistance. Such stable memory must not be easily overwritten by accidental operation of a common or garden variety of program bug. (We are not considering here the rarer and more malignant forms of bug.) It might well be possible to encode information written to stable memory by encryption or translation to unforgeable numbers, but it would be simpler and cheaper for the writers to memory to present a valid token. Such a token could well be the previous contents of the storage location. The "compare and rewrite if correct" operation would have to be atomic, and therefore either locked by software, presumably at the transaction level, or perhaps by microcode, at the memory bus level. A further application of this idea is obviously in optimistic locking protocols for databases.

Probably none of these ideas is new; indeed, it would surprise us if they were. We offer them only as support for our belief that mechanisms which have been extensively investigated by workers in the security field are the same mechanisms as those used where reliability is the main language that is spoken.

It is perhaps appropriate to end this discussion of security, and its close parallels to reliability, by returning to the comment made above about the apparent inconsistency between the present arguments and the comments, contained in the Clearwater Beach paper cited earlier, about the merits of a careful separation between security and reliability. The distinction to be made is, of course, between (i) security and reliability as characteristics or qualities, and (ii) particular security and reliability mechanisms. It is entirely natural to find that similar characteristics demand similar mechanisms. However it is also often highly advantageous to implement distinct mechanisms (whether concerned with reliability or security) that do not need to affect a system's visible functionality, quite independently of each other.
5. REALITY

When you get interesting analogies like the ones we have been exploring between reliability and security, the obvious question to ask is "Why?". Is there an underlying single phenomenon which is being manifested in different ways? Our obvious answer, so far, has been simply to say that security like reliability, is just a subdivision or subset of dependability; a security specification is just another form of specification and hence the mechanisms for ensuring that such a (security) specification is met are just a subset of the more general dependability mechanisms that are available.

This answer is, however, rather unsatisfactory since it is really no more than a re-statement of the analogy at the same level of abstraction; it gives no new or deeper insight. It is here that we re-emphasise our insistence on the importance of considering systems as embedded in the larger system which constitutes the outside world, since the analogy has its roots in the real world; we can appeal to the real world principle or constraint that "Things Go Wrong" and cite as a particular instance of Things, people's sense of honesty and discretion. One can attempt to deal with or at least limit the problems imposed by Murphy's Law by means of fault prevention and/or fault tolerance. Particular mechanisms are known in each category. All that we have done is to interpret known fault tolerance mechanisms in the particular context of security problems.

It is natural to ask, what other relevant general laws of nature are there, similar to Murphy's Law. Here are a few:

- A system can always be seen as part of a larger system.
- Distance means time.
- Two things cannot be in the same place at the same time.
- In order to refer to a thing, it needs a name.

As will be seen, some of these laws are scientific laws of nature, some are pseudo-(or proto-)scientific laws of human experience. The distinction is probably unimportant. What is important is that they represent very broad constraints on a system. (Roughly speaking, the breadth of a constraint is inversely proportional to the probability that an arbitrary system will violate it.)

If you consider the constraints on a system, the narrowest one ideally should be represented in documented form as a complete specification of the system. In practice it usually just corresponds to, that is defines, and is defined by, the system implementation itself. Starting from this constraint, one can envisage a more general and hence broader set of constraints. For example, for an embedded microprocessor system one might have:

- The software shall occupy <as much memory as it does>
- The software shall not occupy significantly more memory than is the case in competitive devices
- The software shall use industry-standard memory chips
- You cannot get a quart into a pint pot!

For any set of constraints on a system, one can define a partial ordering, where C1 > Cj holds if constraint C1 could not be violated without violating constraint Cj. We can take as the widest possible constraint, that the system must exist in the real
world. We thus have a partial ordering, a lower bound (corresponding to the system's implementation), and an upper bound (corresponding to the real world) — and hence a lattice of constraints.

What does this lattice tell us? Ideally, it helps us to explore the space of potentially useful system designs. However, the curious thing is that the higher up we go, the less well we understand the constraints. We can take as an example our previous general observation that things need names in order to be able to refer to them. A lot is known about ingenious data structures for rapid look-up in a name server; something is known, but not quite so much, about distributed name interpretation; there is no decent coherent theory of naming and binding which is abstract enough to encompass and enlighten all examples of these issues in computing; and philosophers have spent man-aeons arguing really interesting questions like "What is a name?". Other examples of high-level issues where a deep theoretical understanding is lacking are concurrency, error handling, distribution and communication.

What we do is try to understand the nature of those constraints nearest the bottom of the lattice, for perfectly good reasons of pragmatics and limitations of time and ability and so on. Now and then perhaps we glimpse some partial insight into the higher levels of the lattice and then the mist comes over again and it is back to toiling over the specification of an alternating bit two phase Byzantine protocol in CCS, or whatever.

One difficulty in trying to construct a detailed constraint lattice is that we may at some level formulate a constraint that is actually in conflict with a higher constraint. As an example of a specification which, it can be argued, is in conflict with a higher level requirement, we take "This real-time fault-tolerant distributed system must be implemented in ADA". The conflict arises from the fact that ADA does not address the issue of fault-tolerant programming in distributed systems[15]. In particular, there is no defined means for the run-time system to inform a program of processor failure or network partitioning, nor does the language appear to have the facilities to enable a program to detect such occurrences itself. (A further technical defect is that the semantics of timed and conditional entry calls, which could (with some ingenuity) be used for this purpose, are defined in such a way as to give rise to inconsistencies in a distributed environment with non-zero transit delay.)

One use that can be made of the lattice of constraints is in explaining to a client the nature of the incompatibilities between his conflicting set of requirements. One is often faced with the task of helping a client decide the minimum set of constraints that must be relaxed in order to begin to be able to design a system that meets the (revised) requirements. Experience shows that many clients have an extraordinary one-level view of their initial set of requirements and that an important task of the system designer is to impose some structure on that one-level view. Another feature of the lattice is that it can be used to provide a (sometimes post hoc) justification of a particular design. UNIX United and the Newcastle Connection provides a good example. This was initially implemented by discovering how to intercept system calls surreptitiously and divert them to some other machine. Later, we discovered a number of interesting ramifications in the areas of security and reliability which exemplified more general principles of separation and replication; and then we elucidated a set of system structuring principles which encompassed not only UNIX United and its various applications but many other things besides.
What might a top-down approach to a reliable, secure design look like? It would not start with an information theoretic algebraic model on which, with great difficulty, theorems were proved and specifications were verified, and the verifications were (again with great difficulty) themselves verified, and so on, until ultimately "This system is secure" was ceremoniously stamped on it. Rather, our knowledge of system structuring principles would force us to pose such questions as: what are the biggest blocks in the system, and are they so constructed that they could be made part of some larger system? Our knowledge of system reliability considerations would force us to consider what we would do if despite all the proofs of impenetrability we actually found, so to speak, a spy in the building with the anti-gravity booster blue-prints in his coat pocket. Shoot him? (A better understanding of reliability would suggest: make him put them back in the safe, and then shoot him!)

Although these are all examples of systems thinking at play, there is a more serious point lurking behind them, and that is best expressed in terms of a meta-model, or model of the modelling process. In this model of the thinking process, there are three stages. In the first stage, selected elements of the real world are translated into symbols (or concepts). This is an abstraction process which yields a set of primary symbols. In the second stage, new symbols are derived from the primary symbols by a cogitative process or combination of processes, such as refinement, elaboration, calculation, reasoning and symbol manipulation. In the third stage, the derived symbols are re-translated back into the real world by an interpretative process of construction, instantiation, or explanation.

It seems to us that the most interesting and mysterious process in this model is the first step, that of abstraction from the real world. Too often this crucial step is abbreviated or performed in a slipshod manner. Essentially this is a step of simplification not over or under-simplification and not the wrong one, either. But we do not know any way of achieving the right simplification other than playing around with real world concepts using real world language, and thinking about what we are doing.

6. CONCLUDING REMARKS

So what can we learn from this attempt to step back from the detailed problems of reliability and security. Perhaps the most immediate lessons concern the possible benefits of applying fault tolerance concepts and techniques to security problems, and so are really better aimed at a different audience. However we would claim that an occasional stepping back, such as we have attempted, is often worthwhile. As a profession, we seem to specialise in re-inventing the wheel, and in inventing jargon that, by accident or design, obscures the fact of re-invention. (For example, the prefix "micro" - whether in front of the word "computer" or the word "program" - does not, present appearances to the contrary, create a totally new field, with a totally new set of design issues.) There are more than enough unsolved problems to be tackled, without having to create spurious ones by ignoring known solutions.

But it is not sufficient to regard a real problem as solved when we have a formal solution to an inadequate abstract model of the real problem. Well-directed rigour is one thing, formality for formalism's sake is another. Echoing sentiments expressed by Lamport[16], in a discussion of unsolved concurrency problems, it is important to understand the differences between any model and reality. The systems which we try to build, if they are to benefit the outside world, must conform to its uncomfortable
realities, as well as to our comfortable theories. This is a lesson which is, for example, well known to cryptanalysts, at least of the professional variety, as centuries of history demonstrate. Would that it were as well known to everybody concerned with various current plans for massively sophisticated real time distributed computing systems, the possible consequences of whose unreliability or insecurity are too horrific to contemplate.

7. ACKNOWLEDGEMENTS

Whilst developing the arguments contained in this paper, we have had considerable benefit, as well as enjoyment, from discussions with our Computing Laboratory colleagues, and Tom Anderson in particular, and also with Derek Barnes and Simon Wiseman of the Royal Signals and Radar Establishment (RSRE). The Newcastle Reliability Project is sponsored by the UK Science and Engineering Research Council, and by RSRE.
References


