DEPENDABILITY, STRUCTURE AND INFRASTRUCTURE[1]

Brian Randell

School of Computing Science,
University of Newcastle upon Tyne, UK

Abstract: The role that system structuring techniques plays in helping to cope with system complexity and achieve system dependability is analyzed. This leads on to a discussion of (i) the problems of the various kinds of assumptions, both deliberate and unthinking, that are made by system designers, (ii) the increased difficulties regarding such assumptions that can face the designers of complex systems that are intended to function as infrastructures, and (iii) the problems of defending systems and infrastructures against malicious attacks, including those which cause initially-unsuspected structural damage.

Keywords: Dependability, Information Infrastructure, Structure, Failures, Malicious Faults, Systems-of-Systems.

1. INTRODUCTION

Governments, industry and the public at large are becoming ever more dependent on a global information infrastructure that is made up of countless inter-linked computer systems. Yet this infrastructure can best be described as fragile, rather than dependable. This is due not just to the immense size and complexity of this information infrastructure and hence the number of faults that occurs almost unavoidably. Rather, it is also due to the damage this infrastructure suffers from the activities of a seemingly ever-increasing number of criminals and hackers.

Unfortunately the current less than satisfactory situation is likely to deteriorate as the global information infrastructure expands to encompass large numbers of huge networked computer systems, perhaps involving everything from super-computers and massive server "farms" to myriads of small mobile computers and tiny embedded devices. Yet in many cases such systems will be expected by the governments and by the large industrial organizations that aim to install them, and indeed by the general public that will have to interact with them, to function highly dependably and essentially continuously. This at any rate would be the consequence should various current industrial and government plans –see for example [European

[1] This paper draws on material in Avizienis et al [2004], [Jones and Randell 2004] and [Randell 2000].

Commission 2002]– come to fruition. Such plans have been formulated despite the fact that the envisaged systems, and hence the information infrastructure that they both rely on and help constitute, seem certain to be far more complex than today's global information infrastructure.

2. SOME STATISTICS

The present situation is perhaps best illustrated by the following sample claims and statistics relating to the (un)dependability of the global information infrastructure:

- The average cost per hour of computer system downtime across thirty domains such as banking, manufacturing, retail, health insurances, securities, reservations, etc., has recently been estimated at nearly $950,000 by the US Association of Contingency Planners [2].)
- The French Insurer's Association has estimated that the yearly cost of computer failures is 1.5 –1.8B Euro (10 – 12B Francs), of which slightly more than half is due to deliberately-induced faults (e.g. by hackers and corrupt insiders) [Laprie 1999].
- The July 2003 report on "Government IT Projects" [Pearce 2003] states that in the UK: "Over the past five years, high profile IT difficulties have affected the Child Support Agency, Passport Office, Criminal Records Bureau, Inland Revenue, National Air Traffic Services and the Department of Work and Pensions, among others."
- Networking dependability is if anything decreasing: "The availability that was typically achievable by (wired) telecommunication services and computer systems in the 90s was 99.999% – 99.9%. Now cellular phone services, and web-based services, typically achieve an availability of only 99% – 90%." [AMSD Roadmap 2003]
- The problem of deliberate attacks on networked computer systems, and potentially via them on other major infrastructures, which is already serious is growing rapidly, as evidenced by SEI/CERT's statistics on annual numbers of incident reports: 9,859 (1999), 52,568 (2001), 137,529 (2003).
- Price Waterhouse Cooper's "Information Security Breaches Survey 2002" for the UK Dept. of Trade and Industry states that "44% of UK businesses have suffered at least one malicious security breach in the past year, nearly twice as many as in the 2000 survey". [3]

---

[2] http://www.acp-wa-state.org/Downtime_Costs.doc

[3] http://www.dti.gov.uk/industry_files/pdf/sbsexecsum_2002.pdf

3. DEPENDABILITY

The very concepts of ‘dependability’ and ‘failure’ [Avizienis et al. 2004] need careful definition in order to make sense of the problems that can afflict a system whose very complexity is a major cause of difficulties. For our purposes here the following will suffice:

> **Dependability** is the ability to avoid failures that are more frequent and more severe than is acceptable,

where:

> A system, operating in some particular environment, may fail in the sense that some other system makes or could, in principle, have made a judgement that the activity or inactivity of the given system constitutes **failure**.

The above definitions are based essentially on just two basic concepts, namely ‘system’, and ‘judgement’, for each of which any standard dictionary definition will suffice [4].

The judgemental system is a potential or actual observer of the given system, and may be an automated system, a user or an operator, a relevant judicial authority, or whatever. This judgemental system may or may not have a fully detailed system specification to guide it, though evidently one would expect such a document to be available to guide the system construction task. Ideally such a specification would be complete, consistent, and accurate – in practice there are likely to be many occasions when it is the specification rather than, or as well as, the specified system which is judged as failing.

Different judgemental systems might, of course, come to different decisions regarding whether and how the given system has failed, or might fail in the future, in differing circumstances and from different stakeholders' viewpoints. The crucial issue might, for example, concern the accuracy of the results that are produced, the continuity or timeliness of service, or the success with which sensitive information is kept private. Indeed, it is likely to concern some balanced combination of such attributes. (Thus, especially with highly complex systems, the notion of what constitutes a failure and whether one has occurred can often be quite subjective.)

Moreover, such a judgemental system might itself fail in the eyes of some other judgemental system, a possibility that is well understood by the legal system, with its hierarchy of courts. So, there is a (recursive) notion of ‘failure’, which clearly is a relative rather than an absolute notion. So then is

[4] Another definition of **dependability** is “that property of a computer system such that *reliance can justifiably be placed on the service* it delivers”, which though usefully intuitive begs the questions of what is meant by ‘reliance’, and who or what is providing the ‘justification’.

the concept of dependability, at any rate when one is dealing with hugely complex systems.

It is in fact necessary to distinguish between three basic concepts – 'failure', 'error' and 'fault'.

> An **error** is that part of the system state that is liable to lead to subsequent failure; an error affecting the service that is being provided by a system is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault**.

Note (i) that an error may be judged to have multiple causes (for example the occurrence of an attempted attack on the system, and the existence of an exploitable vulnerability within the system, left there by a failing system design and implementation process), and (ii) an error does not necessarily lead to a failure (for example, error recovery might be attempted successfully and failure averted).

Thus a failure occurs when an error 'passes through' the observation interface and affects the service delivered by the system; a system being composed of components that are themselves systems. The manifestation of failures, faults and errors follows a 'fundamental chain':

... → failure → fault → error → failure fault → ...

Such chains can exist within systems. System failures might be due to faults that exist because, for example, of the failure of a component (i.e. sub-system) to meet its specification.

However, these chains also exist between separate systems. For instance, a failure of a system-design system (constituted by a human design team together with their computer facilities and tools) might result in the wrong choice of component being made in the system that is being designed, resulting in a fault in this latter system that leads eventually to its failure. Also, a system might fail because of the inputs that it receives from some other system with which it is interacting, though ideally it will be capable of checking its inputs thoroughly enough to minimize such failures.

By having separate terms for the three essentially different concepts – 'fault', 'error' and 'failure' – one has a chance of dealing properly with the complexities and realities of failure-prone components, being assembled together in possibly incorrect ways, so resulting in systems which in reality will still be somewhat failure-prone.

Note that with highly complex networked systems (i) there are likely to be uncertainties and arguments about system *boundaries*, (ii) the very complexity of the system (and its specification, if it has one) can be a major problem, (iii) judgements as to possible causes or consequences of failure can be subtle and disputable and (iv) any provisions for preventing faults from causing failures are themselves almost certainly fallible.

The above definitions, in clarifying the distinction in particular between fault and failure, lead to the identification of the four basic means of obtaining and establishing high dependability, namely:

- *fault prevention* - prevention of the occurrence or introduction of faults, in particular, via the use of rigorous design methods;
- *fault tolerance* - the means of delivery of correct service in the presence of faults;
- *fault removal* - that is, verification and validation, aimed at reducing the number or severity of faults; and
- *fault forecasting* - system evaluation, via estimation of the present number, the future incidence and the likely consequences of faults.

With less than perfect (i.e. real) systems, in the absence of (successful) fault tolerance, failures may occur and will presumably need to be tolerated somehow by the encompassing (possibly socio-technical) system. However, this notion of 'failure tolerance' is just fault tolerance at the next system level.

Note, however, that one of the most important lessons from work over many years on system dependability is the fact that fault prevention, removal and tolerance should not be regarded as alternatives, but rather as complementary technologies, all of which have a role to play, and whose effective combination is crucial – with fault forecasting being used to evaluate the degree of success that is being achieved.

4. ASSUMPTIONS, ASSUMPTIONS, . . .

System designers *always* make (possibly arbitrary or unthinking) assumptions concerning, for example: (i) the types and relative frequencies of the faults that might occur, (ii) the effectiveness with which the various dependability means are being deployed, (iii) the relative importance of various (somewhat antagonistic) dependability attributes: reliability, availability, integrity, security, etc., and (iv) the accuracy of any dependability predictions.

The problems of preventing faults in systems from leading to system failures, and of minimising the severity of failures that do occur, vary greatly in difficulty depending on these assumptions. For example, one might choose to assume that (i) operational hardware faults can be cost-effectively masked (i.e. hidden) by the use of hardware replication and voting, and (ii) any residual software design faults can be similarly masked by the use of design diversity, i.e. using N-version programming, provided of course that the assumption is correct. In such circumstances error recovery is not needed. In many realistic situations, however, if the likelihood of a failure is to be kept

within acceptable bounds, error recovery facilities will have to be provided, in addition to whatever fault prevention and fault masking techniques are used.

The notion of coverage is of relevance to the problem of justifying any such assumptions. In its general sense, **coverage** refers to a measure of the representativeness of the situations to which a system is subjected during its analysis (e.g. in the case of software by means of code inspection, static checking, and program testing) compared to the actual situations that the system will be confronted with during its operational life. This notion of coverage may be made more precise by indicating its range of application, e.g., coverage of a software test with respect to the software text, control graph, etc., coverage of an integrated circuit test with respect to a fault model, coverage of fault tolerance with respect to a class of faults, and, most significantly, coverage of a fault assumption with respect to reality.

By way of an example of the importance of the choice of fault assumptions, consider the problems of error recovery in decentralised systems. If the designer of a distributed database system disallows (i.e. ignores) the possibility of undetected invalid inputs or outputs, the errors that have to be recovered from will essentially all be ones that are wholly within the system. In this situation backward error recovery (i.e. recovery to an earlier, it is hoped error-free, state) will suffice, and be readily implementable, such is the nature of computer storage. If such a system is serving the needs of a set of essentially independent users, competing against each other to access and perhaps update the database, then the now extensive literature on database transaction processing and protocols can provide a fertile source of well-engineered, and mathematically well-founded, solutions to such error recovery problems [Gray and Reuter 1993].

However, the multiple activities in a decentralised system will often not simply be competing against each other for access to some shared internal resource, but rather will on occasion at least be attempting to co-operate with each other, in small or large groups, in pursuit of some common goal. This will make the provision of backward error recovery more complicated than is the case in basic transaction-oriented systems. And the problem of avoiding the "domino effect" [Randell 1975], in which a single fault can lead to a whole sequence of rollbacks, will be much harder if one cannot disallow (i.e. ignore) the possibility of undetected invalid communications between activities.

When a system of interacting threads employs backward recovery, each thread will be continually establishing and discarding checkpoints, and may also on occasion need to restore its state to one given in a previously established and still-retained checkpoint. But if interactions are not controlled, and appropriately co-ordinated with checkpoint management, then the rollback of one thread to prior to a communication will necessitate that the other thread involved in that communication also rollback, and can possibly

result in a cascade of rollbacks that could push all the threads back to their beginnings, i.e., cause the domino effect to occur. (Compare the differing severity of threads T1 and T2, in Figure 1, having to go back to an earlier checkpoint.)

However, the domino effect would not occur if it could safely be assumed that each thread's data was fully validated before it was output, i.e. was transmitted from one thread to another. (Similarly, the effect would be avoided if a thread could validate its inputs fully.) Such an assumption is in effect made in simple transaction-based systems, in which outputs from the database thread to a user thread are allowed to occur only after a transaction has been "committed" in response to a user command. Moreover, in such systems the notion of commitment is regarded as absolute, so that once the commitment has been made, there is no going back, i.e. there is no provision for the possibility that a database output (or a user input) was invalid.

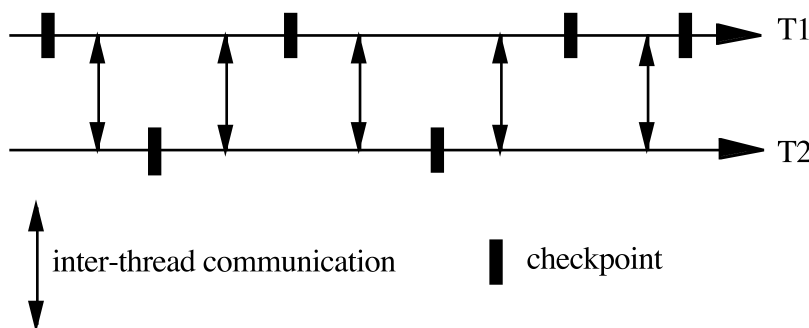

Figure 1: The domino effect

The notion of nested transactions can be used to limit the amount of activity that has to be abandoned when backward recovery (of small inner transactions) is invoked. However, this notion typically still assumes that there are absolute "outermost" transactions, and that outputs to the world outside the database system, e.g. to the users, that take place after such outermost transactions end must be presumed to be valid.

Now let us explore the consequences of a less restrictive scenario. The conversation scheme [Campbell and Randell 1986] provides a means of co-ordinating the recovery provisions of interacting threads so as to avoid the domino effect, without making assumptions regarding inter-thread output or input validation. Figure 2 shows an example where three threads communicate within a conversation and the threads T1 and T2 communicate within a nested conversation. Communication can only take place between threads that are participating in a conversation together, so while T1 and T2 are in their inner conversation they cannot damage or be damaged by T3.

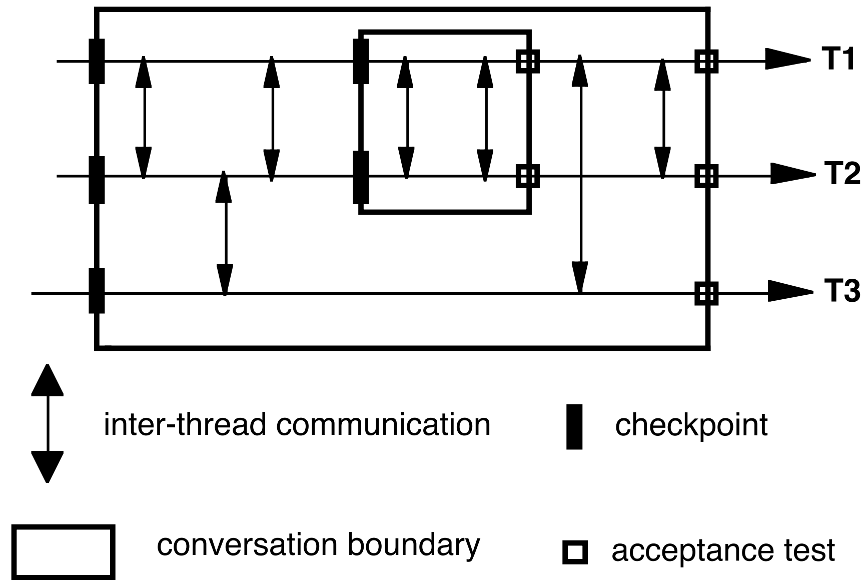

Figure 2: Nested conversations

The operation of a conversation is as follows: (i) on entry to a conversation a thread establishes a checkpoint; (ii) if an error is detected by any thread then all the participating threads must restore their checkpoints; (iii) after restoration all threads then attempt to make further progress; and (iv) all threads leave the conversation together, only if all pass any acceptance tests that are provided. (If this is not possible, the conversation fails - a situation which causes the enclosing conversation to invoke backward error recovery at its level.)

Both transactions and conversations are examples of atomic actions [Lomet 1977], in that viewed from the outside they appear to perform their activity as a single indivisible action. (In practice transaction-support systems also implement other properties, such as "durability", i.e. a guarantee that the results produced by completed transactions will not be lost as a result of a computer hardware fault.) And both rely on backward error recovery.

However, systems are usually not made up just of computers - rather they will also involve other entities (e.g. devices and humans) which in many cases will not be able to simply forget some of their recent activity, and so simply go straight back to an exact earlier state when told that an error has been detected. Thus forward error recovery (the typical programming mechanism for which is exception handling), rather than backward recovery will have to be used. Each of these complications individually makes the task

of error recovery more difficult, and together they make it much more challenging. This in fact is the topic that I and my colleagues have concentrated on these last few years.

Our resulting Co-ordinated Atomic (CA) Action scheme [Xu et al. 1995] was arrived at as a result of work on extending the conversation concept so as to allow for the use of forward error recovery, and to allow for both co-operative and competitive concurrency, i.e. on relaxing the assumptions even further. CA actions can be regarded as providing a discipline, both for programming computers and for controlling their use within an organisation. This discipline is based on nested multi-threaded transactions [Caughey et al. 1998] together with very general exception handling provisions. Within the computer(s), CA actions augment any fault tolerance that is provided by the underlying transaction system by providing means for dealing with (i) unmasked hardware and software faults that have been reported to the application level to deal with, and/or (ii) application-level failure situations that have to be responded to. (However, detailed discussion of this topic is beyond the intended scope of this paper.)

Conversations and CA Actions are methods of structuring the activity of a system, for purposes of error confinement and recovery. The initial issue with regard to system dependability concerns the structuring of the actual system itself.

5. STRUCTURE

A description of a system's *structure* identifies its component systems, their observable boundaries and functions, and their means of interaction. Careful identification of system *boundaries* is fundamental to understanding and distinguishing between failures, faults and errors, so as to analyze possible or actual failure situations and find means of reducing their likelihood, especially in complex systems. System *structuring* thus plays a central role in dependability.

Structure is in effect always "in the eye of the beholder". Having identified a system's boundary, one normally chooses to identify a structuring, in the form of a set of internal boundaries and interactions, which reflects knowledge or assumptions concerning how a system has been or might be constructed, and/or taken apart. Very often the components that are so identified might have had some previous separate existence, and have been chosen from among various possible alternatives. Structuring strategies that facilitate such choices have the additional advantage that they also tend to favour identification of components whose interfaces are seen as relatively simple and/or familiar. However, well-chosen structuring will not only have such characteristics but will also –on grounds of efficiency and effectiveness– identify sets of components that have low coupling and high cohesion. Such

well-chosen structuring greatly facilitates understanding of how a complex system functions.

However, for structuring to have some direct relevance to questions of operational dependability, and in particular fault tolerance, it must be what might be described as strong – **strong** structuring actually controls interactions within and between systems, and limits error propagation in both time and space, i.e. constitutes *real* not just perceived or imagined boundaries.

A well-known form of strong structuring is that provided by the watertight bulkheads in a ship. Clearly, these would do nothing for the ship's seaworthiness, i.e. dependability, if they were constructed merely from cardboard (leave alone if they existed only in the blueprints and not in the actual ship), or if the rules regarding opening and closing of any doors that provided access through them were blithely ignored.

The reality, and hence strength, of physical structure such as provided by bulkheads in ships, or by insulation in electronic systems, can be self-evident. Software structures, as represented by source code constructs such as classes, objects, modules, etc., are more difficult to discern, but nevertheless if retained in some form and used as constraining mechanisms in the operational software can play a similar role in controlling interactions within complex computer systems. (Clearly this involves ensuring that the source code's structuring is not destroyed by, for example, an optimising compiler.)

All this is assuming that the implemented structuring actually matches the system designers' intentions and assumptions regarding system structuring – something that is by no means certain to be the case with systems that are so large and complex as to be beyond any individual designer's comprehension. Needless to say, inaccurate assumptions about a system's actual structure can invalidate any or all coverage assessments that have been estimated on the basis of these assumptions.

An even worse variant of the problem of inaccurate assumptions about a system's structure arises if the original system structure (which perhaps was put in place at least in part as a defence against malicious faults e.g. due to network hackers) has in fact itself been maliciously subverted. At present, the main defences against malicious faults are structuring mechanisms such as firewalls and intrusion detectors, backed up by manual assessment and system recovery actions – actions that may take hours or even days during which the attacked system may not be operational.

A current research priority therefore is that of finding effective automated means of tolerating malicious faults – this involves (i) defending the critical mechanisms, such as the intrusion detection system, against subversion, e.g. by replicating the services or making them difficult for an attacker to locate, and (ii) finding ways of masking or recovering from any errors caused by malicious faults, in order that system operation can be continued without the need for human intervention.

Progress has been made on means for providing a degree of such intrusion tolerance, for example in the MAFTIA (Malicious- and Accidental-Fault Tolerance in Internet Applications) Project [Powell et al. 2001], and by the DARPA OASIS Program[5]. However, it is an open question as to what extent hackers' inventiveness can be defeated by *pre-designed automated* defensive structuring. And one is always left with the conundrum that structuring is both a means of understanding, and of misunderstanding, systems.

6. COMPUTER-BASED SYSTEMS

Most large networked systems embody human beings as, in effect, system 'components', alongside the hardware and software components, though as indicated above such humans may become actively involved only when things go wrong with the computers and their communication. In other cases, there may be a deliberate decision to sub-divide some overall system task into activities that can be readily automated, and those that are best left to human beings to carry out. Systems that incorporate human 'components' in either of these ways are termed here **computer-based systems**.

The various types of components of such systems have rather different failure characteristics: (i) hardware suffers mainly from ageing and wear, although logical design errors do sometimes persist in deployed hardware (see, for example, [Avizienis and He 1999]), (ii) it is mainly software that suffers from residual design and implementation errors, since this is, quite appropriately, where most of the logical complexity of a typical system resides, and (iii) human beings (users, operators, maintenance personnel and outsiders) can at times, through ignorance, incompetence or malevolence, cause untold damage. (Humans also can, with experience, compensate very effectively for computer system faults and failures.) Malevolent activities can range from acts of petty vandalism by amateur hackers, through the efforts of expert and highly devious virus creators, to well-resourced and highly sophisticated attacks by terrorists and enemy states. As indicated earlier, these constitute an increasingly serious problem.

The successful design and deployment of any complex system that interacts directly with humans thus calls for socio-technical as well as technical expertise. One particular problem is that of how best to partition an overall task between humans and computers so as (i) to reduce the probability of failures due to misunderstandings across the human-machine interface, and (ii) to make best use of the greatly differing abilities that humans and computers have with respect to following complicated detailed sequences of

[5] http://www.darpa.mil/ipto/Programs/oasis/index.htm

instructions, and recognising that a situation is both novel and potentially dangerous.

Achieving *predictable* dependability from sophisticated computer-based systems is therefore a major challenge, and a cynical summary is: if you automate a successful complex human-run system, the best you can possibly achieve is fewer but bigger failures. (An obvious example is a complex payroll system – manual payroll systems produce lots of inaccurate cheques, but none has ever generated one for £999,999,999.99.) Such large failures may be due to careless implementation, but the most difficult to avoid are those that are due to mistaken design assumptions. Yet many of the complex systems that are now being planned are computer-based systems, that automate many functions that were previously carried out manually, with all that this implies regarding the difficulty of achieving a high degree of continuity of operation of the global information infrastructure.

7. SYSTEMS OF SYSTEMS

Large systems of any kind are rarely constructed *de novo*, but rather built up from earlier systems. Indeed, they are often actually constructed out of multiple pre-existing operational systems; hence, the term **systems-of-systems**. Moreover, there is almost always a requirement for large systems to be capable of being adapted so as to match changes in requirements – due to changed functionality and changes in system environments – and there is likely to be an increasing need for systems that can themselves evolve. This is needed, for example, in order to serve a dynamic environment constituted by large numbers of mobile devices moving into and out of range.

The typical process of constructing a system-of-systems out of systems that were not designed with this need in mind can best be described as involving "graunching"[6]. This is despite the existence of various more-or-less sophisticated tools and techniques, e.g. such software engineering mechanisms as brokerage services and wrappers. But the resulting systems-of-systems, however put together, tend to suffer from their own special dependability problems.

This is particularly the case when the individual systems are separately managed, and continue to function more or less independently, each with its own individual goals. In such circumstances, management decisions (e.g. regarding system shut-downs or changes) can in effect constitute an additional source of faults.

Another new, or at least exacerbated, class of faults is that of mismatches at the interfaces between component systems. These are typically

[6] To *graunch* (v.t.): to make to fit by the use of excessive force [WW2 RAF slang]

semantic (e.g. confusion between Fahrenheit and Celsius) not just syntactic, or due to fundamental policy incompatibilities. (There is in many quarters a present assumption that a middleware infrastructure facilitating the use of Web Services, XML, etc., will largely solve the problems of creating systems-of-systems, but such facilities mainly help deal with syntax issues.)

But there is also a totally insidious dependability problem caused by the fact that systems-of-systems may come into existence essentially accidentally. Interconnections may be made between systems, without due regard for the consequences, and have the result of creating an accidental system-of-systems, indeed one of such utility it comes to be depended upon, even though there is no-one responsible for ensuring its dependability. And indeed this in part is exactly what has happened on a massive scale, as various separate critical national infrastructures (e.g. for energy, water, transportation, finance, information, etc.) have become much more inter-dependent than anyone had planned – hence the setting up of various national and international investigations into critical infrastructure interdependencies.

8. INFRASTRUCTURE DEPENDABILITY

The term **infrastructure** in general refers to an *underlying* set of *general purpose* facilities. It is a matter of viewpoint whether these facilities are regarded as a system or an infrastructure, since one organization's *system* often becomes another organization's *infrastructure.*

An infrastructure has connotations of being reusable by different individuals/organizations for different purposes on different occasions. Typically, not all of these uses are known to, or even the concern of, the designer(s) of the infrastructure, who therefore must create something that will respond to and support types of use that have not yet been conceived. Moreover, infrastructures typically need to be *capacity engineered* – so that the amount of resource can be changed to meet current and expected demand. (Over-deployment endangers the supplier, under-deployment frustrates the user.)

All these uncertainties about the uses to be made of, and the demands to be placed on, an infrastructure exacerbate the problems of designing for dependability. Design decision-making often has to be based on very vague evidence and statistics, and very suspect assumptions, much more than is the case with systems that are designed for an identified owner and set of users.

Evolvability and adaptability are usually crucial, but an additional dependability challenge. A particular problem is that, just as with ordinary systems, whatever analysis was undertaken in order to establish that an infrastructure is likely to be adequately dependable, may have to be redone virtually from scratch for a modified version of the infrastructure. (Ideally the

amount of reworking of the dependability analysis would be commensurate with the extent of the change.)

Given the uncertainties as to the amounts and types of demand that an infrastructure may have to face, decisions involving trade-offs between the various attributes of dependability (e.g. between security and availability) are likely to be uninformed initially. In essence the problem is that infrastructure dependability also needs to be capacity engineered – and means provided of making effective trade-offs between the various dependability attributes, as their relative importance changes. This can be a major challenge.

9. INFRASTRUCTURE AND STRUCTURE

The 'Idealized Fault-Tolerant Component' diagram (see Figure 3) is a simple, indeed simplistic, structuring technique that shows one approach to distinguishing between various sorts of system interactions, in particular identifying and classifying those that relate to system activity aimed at error recovery.

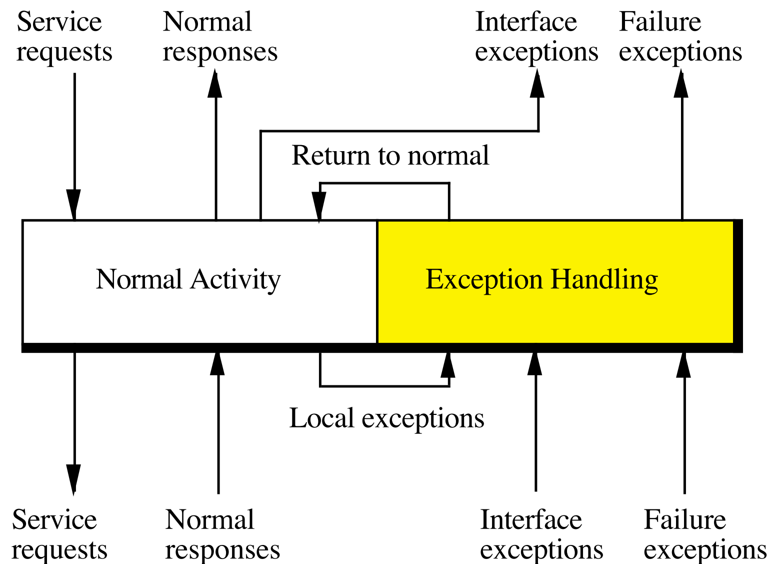

Figure 3: Ideal Fault-Tolerant Component

In fact the diagram relates to the "failure → fault → error" causal chains that can occur, either between systems and their components, or between interacting (though otherwise separate) systems. The terminology used is that of programmed exception handling, even though it can equally well relate to inter-system interaction protocols. It implies that is useful to

distinguish carefully between normal processing, and that involved in error recovery, and between normal and erroneous activations and activities of systems or system components.

Given that an infrastructure is just a special kind of system, or rather a system viewed from a particular vantage point, the diagram can be used in connection with certain types of "failure → fault → error" chain between an infrastructure and the systems the infrastructure is supporting, namely chains related to failures in the systems being supported, or by infrastructure *service* failures.

But we typically use the term 'infrastructure' for a system that is *supporting,* in the sense of initiating and/or enabling the continued effective existence of, some other system, not just interacting with it. This being the case, one can envisage a class of infrastructure failures that in effect destroy the supported systems or their means of communication (e.g. an electrical power blackout, a network outage, or a middleware crash). This class, which includes the case of one system being used to design or construct another, forms the third type of relationship across which "failure → fault → error" chains can operate (the others being system interaction relationships system and composition relationships) – and is one for which a richer more dynamic structuring concept diagram still has to be developed.

10. CONCLUDING REMARKS

The main message of this paper is that when one is considering complex systems (and infrastructures), it is important to recognise that all the relevant concepts and entities – system, dependability, failure, fault, error, failure, structure, and infrastructure – are always to be understood as being relative to some current viewpoint, rather than having some fixed objective meaning. (Indeed even the notion of complexity, which is often spoken of as though it is a system property, in also relative, in that whether something is regarded as complex or simple depends on one's knowledge and experience.)

Moreover, every viewpoint implies a set of assumptions – not all of which will be recognised as such, despite efforts to provide high levels of assumption coverage. During system design, there may be differing assumptions as to the types of demand that will be placed on a system, and the value that will be placed on different system attributes, e.g. usability and security. The whole process of debugging a complex system involves the difficult task of identifying and often repeatedly modifying one's assumptions as to how the system actually works and what possible faults would explain the failure symptoms, etc. And in complex situations different people may, quite understandably, come to different (equally defensible) conclusions, and hence propose alternative ways of repairing the system.

In particular, there will be assumptions about system structure, and these can be the most difficult ones to recognise and allow for. For example, a hacker who is trying to penetrate a system and obtain confidential data will by means of probes and experiments be building up one body of knowledge and assumptions about how the system is structured, in particular how its defences are organised. A system administrator will, through experience of the system, access to its documentation, etc., have a perhaps very different and much more detailed set of information about how the system is (assumed to be) structured. (And the hacker may well have found, or indeed managed to insert, vulnerabilities (i.e. faults) into the system, which in effect – did he or she but know it – completely invalidate the system administrator's information and assumptions!)

Complexity is the main enemy with regard to achieving dependable systems and infrastructures, and structure is one of the main means of mastering this complexity. In particular, good system structuring helps one to deal with the unavoidable additional complexity that results from adequately realistic fault assumptions.

But it would seem that, particularly when one has to allow for malicious attacks by intelligent adversaries on a system's or an infrastructure's protective structuring, there are at least in principle some fundamental limitations as to what can be achieved solely by automated defences – assuming, surely rather reasonably, that these automated defences cannot be made to function with a level of intelligence (and deviousness) comparable to that of their human adversaries.

11. ACKNOWLEDGMENTS

The material in this paper has benefited from numerous discussions with many colleagues, both at Newcastle and in IFIP Working Group 10.4. This particular text has been very helpfully commented on by Peter Andras.

12. REFERENCES

[AMSD Roadmap 2003] AMSD Roadmap. *A Dependability Roadmap for the Information Society in Europe,* Accompanying Measure on System Dependability (AMSD) IST-2001-37553 - Work-package 1: Overall Dependability Road-mapping. Deliverable D1.1, 2003.

[Avizienis and He 1999] A. Avizienis and Y. He. "Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors," in *Proc. Dependable Computing for Critical Applications (DCCA '99),* pp. 3-23, San Jose, CA, 1999.

[Avizienis et al. 2004] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp.11-33, 2004.

[Campbell and Randell 1986] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Software Engineering*, vol. SE-12, no. 8, pp.811-826, 1986.

[Caughey et al. 1998] S.J. Caughey, M.C. Little and S.K. Shrivastava. "Checked Transactions in an Asynchronous Message Passing Environment," in *1st IEEE International Symposium on Object-Oriented Real-time Distributed Computing,* pp. 222-229, Kyoto, 1998.

[European Commission 2002] European Commission. *e-Europe 2005 Action Plan: An Information Society for All,* COM(2002) 263 final, European Commission, 2002.

[Gray and Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and techniques,* Morgan Kaufmann, 1993.

[Jones and Randell 2004] C. Jones and B. Randell. *Dependable Pervasive Systems,* Technical Report No. 839, School of Computing Science, University of Newcastle upon Tyne, 2004.

[Laprie 1999] J.-C. Laprie. "Dependability of Software-Based Critical Systems," in *Dependable Network Computing, ed. D. R. Avresky*, Kluwer Academic Publishers, 1999.

[Lomet 1977] D.B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions," *ACM SIGPLAN Notices*, vol. 12, no. 3, pp.128-137, 1977.

[Pearce 2003] S. Pearce. *Government IT Projects,* Report 200, Parliamentary Office of Science and Technology, 7 Millbank, London, 2003.

[Powell et al. 2001] D. Powell, A. Adelsbach, C. Cachin, S. Creese, M. Dacier, Y. Deswarte, T. McCutcheon, N. Neves, B. Pfitzmann, B. Randell, R. Stroud, P. Veríssimo and M. Waidner. "MAFTIA (Malicious- and Accidental-Fault Tolerance for Internet Applications)," in *Supplement of the 2001 Int. Conf. on Dependable Systems and Networks,* pp. D32-D35, Göteborg, Sweden, IEEE Computer Society Press, 2001.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp.220-232, 1975.

[Randell 2000] B. Randell, "Facing up to Faults (Turing Memorial Lecture)," *Computer Journal*, vol. 43, no. 2, pp.95-106, 2000.

[Xu et al. 1995] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud and Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," in *Proc. 25th Int. Symp. Fault-Tolerant Computing (FTCS-25),* Los Angeles, IEEE Computer Society Press, 1995.