ABSTRACT

We discuss a general approach to the design of fault-tolerant computing systems, concentrating on issues of system structuring rather than on the design of particular algorithms. Three forms of structuring are described. The first is based on the use of what we term "idealized fault-tolerant components". Such components provide a means of system structuring which makes it easy to identify what parts of a system have what responsibilities for trying to cope with what sorts of fault. The second is a "recursive structuring" scheme which involves using complete computers as the basic idealized fault-tolerant components of a distributed computing system whose functionality matches that of its component computers. Finally we discuss a generalization of the usual concept of an "atomic action", which provides a means of structuring both forward and backward error recovery in distributed systems. These discussions are given in general terms, and also illustrated by brief accounts of recent and current work at Newcastle on the construction of UNIX-based fault-tolerant and distributed systems.

1. INTRODUCTION

The most straightforward way of constructing reliable computing systems would be to use only reliable components, and to put them together only in accordance with correct designs. In practice one often has to try to achieve reliability despite the unreliability of the hardware and software components used. Moreover (though this is less often admitted) one may well not be able to guarantee that the overall system design is absolutely faultless. Thus strategies aimed at fault avoidance or removal (prior to use of a system) must usually be complemented by strategies aimed at tolerating the presence of faults.

In this paper we will discuss a general approach to the design of fault-tolerant systems. Our approach concentrates on issues of system structuring rather than on the design of particular algorithms. This is because, with computing systems that have to meet complex and demanding specifications, the overall reliability levels achieved will depend crucially on the extent to which the system design can be kept simple. Thus, in our view, careful structuring is at least as important as are clever algorithms to the achievement of successful fault-tolerant system design.

This viewpoint has motivated research over the years at Newcastle, and has led us to a style of system design which is based on what we term "idealized fault-tolerant components". Such components provide a means of system structuring which makes it easy to identify what parts of a system have what responsibilities for trying to cope with what sorts of fault. Moreover, by taking complete computers as the basic idealized fault-tolerant components, one can make use of a "recursive structuring" scheme which simplifies many design issues. However these schemes of structuring the software and hardware comprising a system need to be used in such a way as to achieve an appropriate structuring of the complex asynchronous activities to which the system can give rise, in particular those related to fault-tolerance. In common with other groups, we have been investigating the use of so-called "atomic actions" for this purpose.

These three forms of structuring are described in Sections 2, 3 and 4 below. The discussion is given in general terms, and is illustrated by brief accounts of recent and current work at Newcastle on the construction of fault-tolerant and distributed systems based on the UNIX* system.

2. IDEALIZED FAULT TOLERANT COMPONENTS

Systems, and their components, can be regarded as performing operations in order to provide responses to requests. Within a system in which it is acknowledged that faults might exist these faults can, from the viewpoint of a given component, be grouped into three categories:

(1) faults within the component itself,

(2) faults in the sub-components or co-existing components that a component makes use of, and

(3) faulty requests made of the component by its environment, i.e. the enclosing component or the co-existing components with which it is

*UNIX is a Trademark of Bell Laboratories.
interacting.

Potentially therefore, in a system which is intended to be fault-tolerant, each component should be designed to deal appropriately with each of these three very different situations. Ideally a component should seek to mask its own faults, and any unmasked faults in the components that it makes use of, so that it can appear fully reliable to its environment. In general, however, this will not always be possible. Thus each component should have pre-defined means of reporting to its environment that a fault has occurred, if it has been unable (or has not been designed) to mask the fault.

On the other hand a component cannot be expected to mask the faults (in its environment) that cause the component to be requested to perform an operation which is outside its specification. (A component which is designed, say, to calculate the square root of its input cannot be expected to produce a real result from a negative input.) However we would again argue that the component should have a defined means of reporting the problem back to its environment, where attempts to mask the fault would be appropriate.

Our notion of an "idealized fault-tolerant component" is concerned with these issues of fault reporting, and of the assignment of responsibility for attempts at fault masking. In this section we deal just with a single component and its interactions with its environment, deferring until Section 4 consideration of the structuring of systems constructed out of multiple components, and so capable of asynchronous behaviour.

What is significant about an idealized fault-tolerant component is that it implies a scheme for structuring systems which incorporate various means of tolerating various sorts of faults. The structuring scheme makes minimal assumptions about what sorts of fault cannot occur, and what sorts of fault masking will be achieved - the one requirement is, naturally, that faults do not invalidate the planned structure of components and their inter-relationships.

The scheme requires that, in general, each component should have identifiable, and in principle separate, means for dealing with the above three categories of fault. In hardware terms, we are in part arguing for a design based on self-testing components [1], since it is each component's responsibility to alert its environment when it cannot carry out a requested operation. In programming terms, our scheme is in fact a suggested discipline for exception handling. It is illustrated in Figure 1, in which local exceptions, failure exceptions and interface exceptions are the respective means by which the three categories of fault listed above are reported.

If a component either receives an abnormal response from an invocation of another component or detects an error or abnormal condition during normal activity, it should raise an exception and invoke appropriate fault-tolerance measures.

Recovery is an abnormal activity of the component and is continued until the component either returns to its normal activities or signals an exception. The relationship between the normal and abnormal activity of a component and the raising and signalling of exceptions is shown in Figure 1. Note that an exception is raised within the component, but signalled between components. The flow of control of a computation within a component should change as the result of a raised exception. Such a modified or exceptional flow of control is distinguished from the normal flow of control. Within a program, exceptional flow of control is associated with code fragments that are called exception handlers.

Exceptions, software components, and exception handlers are associated by a handling context. If the fault tolerance measures provided by a handler are successful, the handler will provide a normal return, from the component which raised the exception, to the component which invoked that component. If the fault tolerance measures are unsuccessful or inadequate, a handler should signal a failure exception. Abnormal control flow continues in an exception handler of the invoking component.

An exception handler is a component and may have its own context, exceptions, and exception handlers. This permits the nesting of exception handling facilities. If an exception is raised within a component (or an exception handler) that does not have a context defining an appropriate handler, the component fails and a failure exception is signalled.

Service Normal Interface Failure Interception Signals Signals
Requests Responses Exception Exception

| | | | | | | | | |

| v | v |

Figure 1: An Idealized Fault Tolerant Component

The scheme is based on the so-called "termination model" of exception handling [2]. In other
words, we require that the operation that a component undertakes for a particular service request be terminated by the provision of a normal or abnormal response to the environment. Thus an operation cannot be resumed after the environment has dealt with an interface or failure exception - the component can only be asked to start another operation.

With such a structuring scheme it is possible, and indeed desirable, to specify the interface between each component and its environment completely. This enables the design of the component to be based on just the interface specification, and so to be undertaken independently of that of its environment, even with respect to issues of fault tolerance. This is a very important advantage. When fault tolerance provisions have not been carefully structured their design can be very complex, to the point where the design itself is a source of unreliability.

One of the important tasks of a system designer using this structuring scheme will be to decide what sort of faults can and cannot occur, and what sorts of fault masking can be presumed to be successful. This is so as to determine what exceptions must be defined and what exception handlers must be provided. For example, a component using triple modular redundancy techniques to mask sub-component faults should in principle still have a way of reporting a failure exception to its environment for use if, say, no two of the triplicated components are in agreement. However in some circumstances it might well be decided that it is safe to disregard such a possibility, and hence to omit the provision of such an exception, i.e. to modify the specification of the interface between the component and its environment.

2-1. A Remote Procedure Call Protocol

The approach that we, and several other groups, have taken recently to the structuring of local area networking software can usefully be described in terms of idealized fault-tolerant components. Oversimplifying somewhat, the software can be regarded as being structured into two levels of component, with a third underlying level of component in the form of the network hardware. The lower level software component uses the hardware to provide message-passing facilities, which the upper level software uses to provide a remote procedure call facility.

Software for RPC

-----------------------
Software for
Message Passing
-----------------------
LAN hardware

Figure 2: Communications Software Structure

It would have been possible to provide message passing over the local area network by a component which used a sophisticated protocol involving mechanisms for flow control, error correction and acknowledgements, etc., to give "guaranteed" delivery of undamaged and unduplicated messages. In such circumstances the temptation would have been to treat the message passing as being fully reliable, and to omit any provisions for signalling a communications failure exception to the upper level component.

In fact such a protocol would consume considerable processing and storage resources, since a significant amount of state information would have to be maintained about any data transfer in progress, in order to support the detection of a "connection" between processes. To establish, maintain and terminate a connection reliably is rather complex, and a significant number of messages would be needed just for connection purposes [3].

This would be quite appropriate for wide area packet-switching networks which are liable to damaged, lost or duplicate packets. However it is not so appropriate for a local area network, such as the Cambridge Ring, because of the much greater (though still of course less than total) reliability of the underlying hardware. We therefore chose to provide message passing by direct use of the underlying hardware communications facilities, without trying to mask any of their faults. Instead the message-passing component merely signals an appropriate exception to the enclosing remote procedure call protocol software when necessary. (In fact, once an appropriate interface, with adequate exception signalling, has been defined it is practicable to support it by a variety of communications hardware and message-passing protocols. Thus we plan to support exactly the same message-passing interface when we are exactly the same message-passing interface when we are instead of a Cambridge Ring, a wide area packet-switching network together with a connection-oriented protocol [4].)

The upper-level component therefore contains handlers for exception signals indicating that there has been a communications fault which the message-passing component has not masked, perhaps because it did not attempt to do so. This upper level of software, which provides the rest of the system with a remote procedure call facility, in any case has to deal with problems arising from computers crashing, and hence being temporarily unable to respond at all to messages.

The remote procedure call software in fact has responsibility for trying to ensure that remote procedure calls obey an "exactly once" semantics [5], i.e. that computer crashes and consequent re-sending of messages do not cause unintended repeated execution of a procedure. Being able to do this, it can also readily cope with failure exceptions from the message-passing component. This choice of structure (which is based on the so-called "end-to-end argument" [6]), has proved very satisfactory, since our "light-weight" remote procedure call protocol has proved to be as reliable as, and considerably more efficient than, an existing connection-oriented protocol for the Cambridge Ring [7].
3. RECURSIVELY STRUCTURED SYSTEMS

The notion of an idealized fault-tolerant component can be applied to many different aspects and levels of computing system design. An allied structuring technique that we have been investigating is aimed at the situation where one is taking complete computers as basic system components, and using them to construct a distributed computing system. This "recursive structuring" technique is expressed in the design rule:

A distributed system should be functionally equivalent to the individual systems of which it is composed.

In other words, one arranges to use, as the principal idealized fault-tolerant components of a computing system, computers whose external interface exactly matches that required of the system as a whole. Thus if the distributed system is to provide facilities for parallel processing, the component computers must also provide (at least the appearance of) parallel processing. More importantly, each component computer's naming facilities (i.e., the means it provides to users for identifying its various constituent objects, such as devices, files, programs, etc.) must be independent of whether the computer is in fact an isolated (i.e., complete) system, or merely a component of some larger system. This characteristic is not common in the world of computing systems, despite the fact that it is well known elsewhere, for example in telephone systems. (The telephone numbers used in a company's internal telephone system 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.)

Specifically, the component computer systems need to support a general "contextual naming" scheme for their various objects. In order for a system to be extensible, it should have means for introducing and entering (and leaving) new naming contexts. Such facilities are reasonably common. What is not so common is a system in which all names are context-relative. However this is essential, because of the requirement that a computer be usable in the same way when it is isolated as when it is within a larger system.

The mechanisms that one uses in order to build the overall distributed system out of the component computers must not affect the functionality of the system - they must, in other words, be transparent to the user of the system. These user-transparent mechanisms can however be designed to have beneficial effects on such aspects of the overall system as its capacity, reliability and security - qualities that we term abilities, so as to distinguish them from the logical functionality of the system.

The most important such mechanism is the one that enables a number of component computers to be linked together. The mechanism in fact provides what is often termed "network transparency", since the users of the distributed system will, in view of the design rule that has been followed, be shielded from all issues of inter-computer communication, networking protocols, etc. Note that this is a rather special form of network transparency since one must be able to repeat the construction process, and build a further large distributed system using several first-level distributed systems as though they were basic component computers. A distributed system which is recursively structured in this way is - by definition - indefinitely extensible, at least in theory.

One can contrast this scheme of making a set of computers look like a single computer with the better known technique of using a virtual machine monitor, or "hypervisor", to make a single computer look like a set of computers. (With a so-called "recursive virtual machine monitor" [6], the subdivision can be performed repeatedly, in close analogy to the idea of repeated construction described above.) Both schemes can in fact be regarded as forms of virtualisation, as can user-transparent mechanisms for providing other so-called abilities, in particular reliability.

In a recursively structured computer system, the possible exceptions signalled by the overall system must be the same as the exceptions signalled by the component computers. However, by the use of redundant component computers, one can largely mask their faults, and hence greatly reduce the frequency with which the overall system has to signal failure exceptions. Thus although no change has been made with respect to functionality, the user-perceived reliability of the overall system can be significantly enhanced. This involves construction of a further virtualisation mechanism (e.g. to perform the synchronization and voting needed for triple modular redundancy) but this mechanism can be designed and constructed independently of the virtualisation mechanism that provides network transparency.

In fact one can envisage enhancing the ability of the overall system by means of a whole set of independently designed virtualisation mechanisms. This clear separation of logical concerns greatly reduces overall system complexity. Indeed the important point about a virtualisation mechanism is that its presence is, in a sense, always optional. Thus one can in principle use any selection of virtualisation mechanisms together, in order to obtain a system with the desired abilities.

3.1. UNIX United

The technique of recursive structuring underlies recent work at Newcastle which has resulted in the implementation of yet another distributed system based on UNIX. The system, which for purposes of description we will call "UNIX United", is a distributed system which is functionally equivalent to a conventional UNIX system running on a single processor. All the standard UNIX facilities, e.g., for protecting, naming and accessing files and devices, for input/output re-direction, for inter-process communication, etc., are applicable without apparent change to the system as a whole. UNIX is a particularly suited to the use of the recursive structuring scheme, because of its very simple yet
general scheme for naming files, devices and commands, in which directories serve as the required contexts.

root '//' -->
  |
  |
user   lib
  |
  |
current working --> brian fred
  |
  |
directory
dirl   b   c
  |
  |
a   b

Figure 3: A Typical UNIX Name Space

Figure 3 shows part of a typical UNIX naming hierarchy. Files, directories, etc., can only be named relative to some known "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 repositioned at some other node in the naming tree, but this position must be specified by a context-relative name. Thus all naming is completely context-relative - there is no means of specifying an absolute name, relative to the base of the tree, say. (The base directory can itself be recognized only by the convention that it is its own parent.) Moreover all other means provided for identifying any of the various kinds of objects that UNIX deals with, e.g. users, processes, open files, etc., can be related back to its hierarchical naming scheme. It is for these reasons that UNIX, in contrast to most operating systems, can be said to support a contextual naming scheme.

This simple and elegant scheme of context-relative naming has been taken advantage of in UNIX United by identifying individual component UNIX systems with directories in a larger name space, covering the UNIX United system as a whole. In Figure 4 we show how a UNIX United system spanning an entire university might be created from the machines in various university departments, using a naming structure which matches the departmental structure. (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.)

<table>
<thead>
<tr>
<th>CS</th>
<th>EE</th>
<th>Maths</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>U2</td>
<td>U1 U2 U3</td>
</tr>
</tbody>
</table>

Figure 4: A University-Wide UNIX United System

The figure implies that from within the Computing Science Department's U1 machine, files on its U2 machine will normally have names starting "/.../U2" and files on the machine that the Electrical Engineering Department has also chosen to call "U2" will need to be identified with names starting "/.../EE/U2". Indeed 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.

The network transparency that UNIX United provides has been implemented merely by inserting a software layer, which we call the "Newcastle Connection", into an otherwise unchanged UNIX system. The positioning of the Connection layer is governed by the structure of UNIX itself. In UNIX all user processes and many operating system facilities (such as the 'shell' command language interpreter) are run as separate time-shared processes. 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 a transparent layer that is inserted between the kernel and the processes. It is transparent in the sense that from above it is functionally indistinguishable from the kernel and from below it appears to be a set of normal user processes. It filters out system calls that have to be re-directed to another UNIX system (for example, because they concern files or devices on that system), and accepts calls that have been re-directed to it from other systems. Thus processes on different UNIX machines can interact in exactly the same way as do processes on a single machine.

Since system calls act like procedure calls, communication between the Connection layers on the various machines uses a remote procedure call pro-
tocol (which is based on that discussed in Section 2.1 above), as portrayed in Figure 5.

```
| User programs, | User programs, |
| non-resident   | non-resident   |
| UNIX software  | UNIX software  |
| Newcastle      | Newcastle      |
| Connection     | Connection     |
| procedure calls| UNIX Kernel    |

UNIX1

UNIX2
```

Figure 5: The Position of the Connection Layer

Various additional UNIX virtualisation mechanisms are being investigated. One that has already been implemented in prototype form makes use of triplicate modular redundancy techniques in order to mask hardware faults. This is designed as an additional transparent software sub-system (the Triplicate Modular Redundancy layer) in each of a number of UNIX machines on top of their Connection layers, as shown in Figure 6. The TMR layer goes on top of the Connection layer because it can then rely on the latter to handle all problems relating to the physical distribution of processes, files, etc. Copies of a conventional application program and its files can then be loaded onto each of three machines and run so that file accesses are synchronized and voted upon. Any malfunctioning computer so identified by the voting is automatically switched out and in due course another switched in to replace it. (Being independent of the Connection layer, the TMR layer could, in principle, be used in a single conventional UNIX system. This would give a sort of "temporal triplicate modular redundancy", with three identical processes running in interleaved fashion on a single processor.)

```
| User | User |
| Processes | Processes |
| TMR layer | TMR layer |
| Connection layer | Connection layer |
| Kernel | Kernel |
```

Figure 6: Hardware Fault Masking

4. ATOMIC ACTIONS

The structuring techniques discussed so far have been concerned with the use of systems as components of larger systems, and in particular with the static structure of the software and/or hardware making up a computer system. We now turn our attention to the closely related topic of the dynamic structure of a system. By this we mean the structure of the perhaps complex activities that the components of the system give rise to when the system is operational. By so doing, we can deal with the issues raised by systems in which components can share sub-components, so that there can be complex asynchronous behaviour, and in which the number of components can change. As always, our aim is to cope with complexity by finding means of treating various aspects of the system design independently of each other, even (or rather especially) with regard to the problems of achieving fault tolerance. In this regard we now concentrate on error recovery techniques, i.e., the design of the exception handling mechanisms in a system built using idealized fault-tolerant components.

Error recovery techniques are aimed at allowing a system to continue operation and to resume normal computation. However, other errors may well have been generated in the interval between the manifestation of a fault, and the moment of error detection. Successful fault tolerance must enable the system to continue to function despite errors propagating during the time interval, which may be lengthy, between the first manifestation of a fault and the eventual detection of an error.

So-called "forward error recovery" aims to remove or isolate specific errors so that normal computation can be resumed [9]. It is accomplished by making selective decisions to a system state. Because recovery is applied to a state that contains errors, forward error recovery techniques require accurate damage assessment (or estimation) [10] of the likely extent of the errors introduced by the fault.

In contrast, "backward error recovery" aims to restore the system to a state which occurred prior to the manifestation of the fault. Using this earlier state of the computation, the function of the system is then provided by an alternate algorithm until normal computation can be resumed [11]. (In practice, the most recent restorable system state which is free from the effects of the fault may be difficult to determine. In order to find an appropriate system state, a search technique may be used involving iteratively attempting recovery from successively earlier restorable states until recovery is successful.) Because backward error recovery restores a valid prior system state, recovery is possible from errors of largely unknown origin and propagation characteristics. (All that is required is that the errors have not affected the state restoration mechanism.)

Forward and backward error recovery techniques complement one another, forward error recovery allowing efficient handling of expected conditions and backward error recovery providing a general strategy which will cope with faults a designer did not - or chose not to - anticipate. As a special case, a forward error recovery mechanism can support the implementation of backward error recovery [12].
If a system has but a single component, and the operations performed (for the environment surrounding the system) by this component can be regarded as having an instantaneous or indivisible effect on the state of the system, the system is said to be sequential. As long as the component, and all of its subcomponents, are not also part of any other system, then the system is isolated. The provision of forward or backward error recovery in such isolated sequential systems can be reasonably easy.

If on the other hand a system has a number of components, of which one or more subcomponents are shared, it will be possible for the system to give rise to perhaps very complex asynchronous and interacting activities. The provision of error recovery will be a much more difficult problem in such circumstances. Nevertheless it is possible to limit this complexity, and hence facilitate the provision of recovery in systems, if one can arrange that the overall activity of the system is appropriately structured.

If an operation provided by a system is carried out by the system's components in such a way that there are no interactions between the set of components and the components of any other system, then we can say the system has given rise to an atomic action. (A more rigorous definition can be found in [3].) If a particular operation is always performed atomically, then we term it an atomic operation or, more commonly, an atomic action — a notion which has much in common with that of a "sphere of control" [14].

There are great advantages to structuring systems out of hierarchies of atomic activities. The temporary creation and then destruction of additional components within an atomic action can be dealt with reasonably easily. If a fault, resulting in error propagation, and subsequent successful error recovery all occur within a single atomic action it is possible to ensure that they will not affect other system activities. Furthermore, if the activity of a system can be organized into atomic actions, fault tolerance measures can be constructed for each of the atomic actions independently. Thus, atomic actions provide a framework for encapsulating fault tolerance techniques within modular components. If all the operations on a system are atomic, then that system is an atomic system. Such systems may be used as components in the design and construction of other, more complex, systems as if their activities were primitive atomic actions. Systems that are designed explicitly so as to synchronize the activities of their components in order to form atomic actions have planned atomic actions. Systems may also give rise to spontaneous atomic activities, i.e. ones that arise fortuitously from the dynamic sequences of events occurring in a system. For the purposes of structuring fault tolerance measures, spontaneous atomic activities are of little value even if they can be easily identified as such.

Planned and spontaneous atomic activities represent the two opposite ends of a spectrum of error recovery techniques, depending on the extent to which explicit constraints are imposed upon inter-process communication. The conversation [15] is an example of a planned atomic activity with which backward error recovery is associated. The chase protocol scheme [16] associates backwards error recovery with a more spontaneous form of atomic activity dynamically determined by the protocol from past patterns of interprocess communication and available fault-tolerance provisions. Other error recovery techniques based on atomic activities that are more spontaneous than those of the conversation but less spontaneous than those of the chase protocol exist. For example, the two phase commit protocol [17] explicitly co-ordinates the activities of components leaving a "transaction" but does not require that the components be identifiable beforehand.

However the conversation, chase protocol and two-phase commit techniques concern themselves just with the use of backward error recovery in asynchronous systems. What is needed is a more general scheme for trying to ensure that a system behaves reliably in the presence of faults. The notion of reliability requires that a system have a specification against which the actual results of invoking its operations can be assessed. When an atomic action is executed, a well-defined state exists at the beginning and termination of its activity (although these states may not necessarily be instantaneously observable). The intended relationship between these states constitutes a specification for the atomic action which is independent of any asynchronous activity inside or outside the atomic action.

Such a specification is needed if general exception handlers are to be designed. (No such specification can be available for spontaneous atomic activities, which therefore can at most be used as a basis for backward error recovery.) Thus atomic actions provide a basis for structuring both forward and backward error recovery in asynchronous systems, provided appropriate exception handling rules can be devised. The following paragraphs describe our attempts to do this.

The raising of an exception within a fault-tolerant atomic action requires the application of abnormal computation and mechanisms to implement the fault tolerance measures. If the recovery measures succeed, the atomic action should produce results that are normally expected from its activation. Atomic actions that explicitly return an abnormal result do so only due to the agreement of all their components. Thus we associate exception handling contexts with atomic actions.

An atomic action may contain internal atomic actions. If an exception is raised within an internal atomic action, then the fault tolerance measures of that internal atomic action should be applied. However, an internal atomic action may signal an exception. This exception is raised in the containing atomic action.

Whether one or several of the components carrying out an atomic action raise an exception, we would argue that the fault tolerance measures
necessarily involve all of the components involved in that atomic action. (The fact that an exception has been detected invalidates the assumptions that the components can terminate normally and collectively provide the appropriate results. If some of the components are not required to change their flow of control to execute fault tolerance measures, they do not interact with the other components and hence should participate in a separate atomic action.)

Depictions of an atomic action in which a component raises an exception and each component of the atomic action changes its flow of control are shown in Figure 7 and Figure 8. (Comparing these figures to that given earlier of an idealized fault-tolerant component, the pair of arrows entering the atomic action corresponds to a service request being made of the system, the system being viewed as a fault-tolerant component containing two sub-components. Similarly, the pair of arrows leaving the atomic action as a result of the provision of either the normal response, or an exception signal, by the system.) Although it is so implied by the diagrams, the activities involved in the start and the finish of the atomic action need not be closely synchronized - more sophisticated implementations are possible, though at the cost of increased complexity of error recovery.

![Diagram of Atomic Action](image)

Figure 7: Successful error recovery in an atomic action

Every component involved in the atomic action responds to the raised exception by changing to an abnormal activity. Each component changes to an exceptional control flow so as to execute a handler for that exception. This handler either returns the component to normal activity or signals a further exception. (The change in control flow of a component that awhile as a result of a raised exception in a sequential system is a special case of the changes in control flow that should occur in an asynchronous system.) In Figure 7, the recovery measures implemented by the exception handlers succeed and the normal control flow of the components is resumed. Figure 8 shows the control flow of the components involved in an atomic action when the exception handlers for the components cannot recover.

It is convenient to restrict signalled exceptions so that each component (or exception handler) involved in an atomic action returns the same exception. The signalling of the same exception ensures that the components agree on the abnormal result that should be returned to indicate the failure of the atomic action. (Note that an exception should be raised if two or more components try to signal different exceptions. The exception handlers for this exception should signal a failure exception.) An exception is raised in an atomic action if one of its internal atomic actions signals an exception. Signalling a single exception from an internal atomic action simplifies the selection of the appropriate exception handlers and recovery measures.

If any of the components of an atomic action do not have a handler for a raised exception then all of the components should signal an atomic action failure. However one will commonly arrange that an implicit handler, providing backward error recovery, is invoked by default, if no explicit handler is given.

![Diagram of Atomic Action](image)

Figure 8: Signalling an exception from an atomic action

Our scheme for using atomic actions to structure both forward and backward error recovery in asynchronous systems is described in greater detail in [18]. This deals with such further issues as the simultaneous raising of different exceptions by different components within an atomic action, and the problems of aborting an atomic action, for example in an attempt to avoid missing an imminent deadline.

4.1. Atomic Actions in UNIX United

Our initial experiments concerning the addition of atomic actions to UNIX United mainly address questions of overall system structuring, rather than the problems of implementing the full generality of the atomic action concept and of exception handling as described above. Thus we are concerning ourselves, at least in the first instance, solely with backward error recovery, in fact just at the level of file usage. It will be possible to nest atomic actions, and asynchronous
activity will be supported, but only internal to an atomic action.

The fact that atomic actions are of equal relevance to users of a multiprogramming system, such as UNIX, as to users of a distributed system, prompts their provision in UNIX United by two separate, albeit related, mechanisms. The first of these effectively adds atomic actions to UNIX itself, in the form of three extra system calls:

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

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

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

This additional software is being implemented in the first instance as a separate layer, which will be interposed between the Connection layer and the kernel, as shown in Figure 9 below.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User Processes, etc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Newcastle Connection, with two-phase commit protocols</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic Action support</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIX kernel</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 9: Provision of Atomic Actions

The second mechanism provides network transparency for these additional calls. It is to be implemented by a modest augmentation of the Connection layer. (In the case of the Discard Recovery Point call, it might well be thought necessary to incorporate a simplified form of two-phase commit protocol, which would involve the provision of another system call "Prepare to Discard Recovery Point" by the Atomic Action layer. This should minimize the risk of having some but not all the component UNIX systems complete their Discard Recovery Point calls. In fact virtually all the facilities required within the Connection layer for two-phase commit already exist, being needed to support some existing UNIX system calls.)

5. CONCLUSIONS

The three forms of structuring that we have attempted to describe and justify here are just a modest generalisation and extension of various current approaches to the design and implementation of distributed and/or fault-tolerant computing systems. Nevertheless, we believe that they provide a surprisingly effective and constructive methodology for the design of such systems.

Certainly our experience with UNIX United provides what we regard as strong evidence for the merits of the first two structuring schemes. As reported in [19], a very useful distributed system, enabling full remote file and device access, was constructed within about a month of starting implementation of the Connection layer. Needless to say, the fact that - due to the transparency of the Newcastle Connection - it was not necessary to modify or in most cases even understand any existing operating system or user program source code was a great help! In only a few months this system had been extended to cover remote execution, multiple sets of users, etc. Distributed UNIX systems based on the Connection, using PDP11, VAX, M68000 or ICL PERQ computers, linked either by Ethernet or Cambridge Ring, are now operational at various sites. Moreover two prototype extensions of the system, for multi-level security and hardware fault tolerance, have been successfully demonstrated, and the design of others commenced.

The third structuring scheme described here, an attempt at extending the conventional concept of an atomic action to cover forward as well as backward recovery, and asynchronous systems, is as yet rather more speculative. A full implementation of the scheme, in all its generality, requires resolution of a number of design issues. In particular, practical systems can only be constructed if suitable notations are developed to express the concept of an atomic action. However we believe that the scheme provides a suitable framework for discussing general error recovery in asynchronous systems, and that it will be useful as a guide to the design and implementation of a variety of more specialised and/or limited error recovery schemes.

6. ACKNOWLEDGEMENTS

The discussion of structuring issues presented in this paper owes much to the work of the author's many colleagues at Newcastle and elsewhere, over a number of years. Recent work specifically on the Newcastle Connection and the Remote Procedure Call Protocol, however, has been mainly in close collaboration with Lindsay Marshall, Dave Brownbridge, Fabio Panzieri, Santosh Shrivastava and Jay Black. The work on error recovery in asynchronous systems has been largely carried out with Roy Campbell, of the University of Illinois, during the sabbatical year that he spent at Newcastle. This research has been sponsored by the U.K. Science and Engineering Research Council and the the Royal Signals and Radar Establishment.

References


190


