Abstract:

Real-time systems often have very high reliability requirements and are therefore prime candidates for the inclusion of fault tolerance techniques. In order to provide tolerance to software faults, some form of state restoration is usually advocated as a means of recovery. State restoration can be expensive and the cost is exacerbated for systems which utilize concurrent processes. The concurrency present in most real-time systems and the further difficulties introduced by timing constraints imply that providing tolerance for software faults may be inordinately expensive or complex. The paper asserts that this is not the case, and proposes a straightforward pragmatic approach to software fault tolerance which is believed to be applicable to many real-time systems. The approach takes advantage of the structure of real-time systems to simplify error recovery, and a classification scheme for errors is introduced. Responses to each type of error are proposed which allow service to be maintained.

Practical Software Fault Tolerance for Real-Time Systems

By

T. Anderson and J.C. Knight

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 169
June, 1981

© 1981 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Real-time systems often have very high reliability requirements and are therefore prime candidates for the inclusion of fault tolerance techniques. In order to provide tolerance to software faults, some form of state restoration is usually advocated as a means of recovery. State restoration can be expensive and the cost is exacerbated for systems which utilize concurrent processes. The concurrency present in most real-time systems and the further difficulties introduced by timing constraints imply that providing tolerance for software faults may be inordinately expensive or complex. The paper asserts that this is not the case, and proposes a straightforward pragmatic approach to software fault tolerance which is believed to be applicable to many real-time systems. The approach takes advantage of the structure of real-time systems to simplify error recovery, and a classification scheme for errors is introduced. Responses to each type of error are proposed which allow service to be maintained.

About the author
Dr. T. Anderson is a Lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.
Dr. J.C. Knight is an Aerospace Technologist at the NASA Langley Research Center, Hampton, Virginia, U.S.A. In September 1981 he will take up an appointment as Associate Professor at the University of Virginia, Charlottesville.
1 Introduction

A great deal of research is currently being performed on techniques for the production of better quality software. This research is particularly important for systems where the consequences of any noncompliance of software with its requirements may be disastrous. The failure of an unmanned space mission could cause the loss of expensive equipment before the successful completion of mission objectives. Of even greater concern is the possibility that human lives could be endangered by the failure of a digital avionics system for a commercial air transport.

Many critical systems operate in real time. This means that inputs may be expected and/or outputs must be generated according to some real-time schedule. For example, an avionics system may send commands to control surfaces every tenth of a second of real time. Of course, the traditional problems associated with software arise in real-time systems. The necessary outputs may be produced when they are needed, but they may be incorrect. However, the requirement for operation in real time presents difficulties over and above those normally encountered. For example, execution of a real-time program may successfully produce the output demanded by the program's specification but fail to do so within the imposed real-time deadline.

There are two approaches to the construction of software which must exhibit behaviour that is highly reliable (that is, complying with its specifications most of the time). Avizienis [2] called these approaches fault intolerance and fault tolerance. Fault intolerance, better
referred to as fault elimination or fault prevention, embraces all the various techniques which try to ensure that software contains no faults. For example requirements definitions, precise specifications, design and programming methodologies, proving, and testing can all contribute to an attempt to eliminate the presence of faults from software. Experience has shown that although the adoption of these techniques can be beneficial, a reduction in the incidence of faults and certainly not their complete elimination is all that can be expected. Fault tolerant software incorporates techniques which attempt to ensure that acceptable service is maintained by coping with the faults which remain despite the use of fault prevention measures. Elements are introduced into a system which, in the absence of faults, could be omitted without affecting the behavior of the system.

This paper discusses the application of fault tolerance techniques to real-time software. Real-time systems are modelled as a set of cooperating sequential processes with constraints on their execution time. Each process corresponds to the execution of a program which is part of the system, and provides some subset of the necessary system outputs. Programs are usually executed periodically and, for a given program, the intervals between initiations are almost always the same length of time. An executive ensures that processes of the system are dispatched at the appropriate time and monitors whether each process completes execution within its allotted time.

The approach to fault tolerance presented here is easily applied to practical real-time systems and does not require special-purpose hardware facilities for its implementation. It is not designed for or
limited to any specific project but suggests techniques which can be tailored for any particular application. Using this approach, systems can be constructed which will continue to provide adequate responses in real time under circumstances where faults in the software would normally cause a loss of service. It will be necessary to construct software above and beyond that which has been traditionally considered sufficient. The additional costs may be substantial but must be regarded as unavoidable, as are the costs incurred from the redundancy necessary to provide hardware fault tolerance. Software which does not include every facility to enhance reliability is unacceptable in the critical applications being considered here.

None of the techniques proposed here are intended to cope with hardware faults (although tolerance to some hardware faults may be obtained nevertheless). Hence, it is assumed that the software is executed on hardware of extremely high reliability — such as the SIFT [24] and FTMP [10] computers. To assist in achieving reliable operation, the hardware should include facilities such as high resolution clocks and memory protection.

The work reported here was motivated by the need for reliability in flight software but the techniques have relevance in other real-time applications where the cost of failure is high; for example, the operation of a nuclear power plant or the monitoring of hospital patients.

Previous work in the area of fault tolerant real-time software has been reported by Campbell, Horton and Belford [3], by Hecht [9] and by Kopetz [13]. It is likely that certain military real-time systems have
made some use of software fault tolerance, but in most cases such systems have not been described in the open literature. One exception is SAFEGUARD [20].

Campbell et al considered a system of independent primary processes which individually must provide their respective service within their own time limits. To guard against a primary process not completing in the required time it was suggested that an alternate process be available for each primary process, which would be able to provide a degraded service in substantially less time than is required by the primary process. The deadline imposed on the primary is such that there is always sufficient time to execute the alternate should this prove necessary.

Hecht has made various specific suggestions for incorporating fault tolerance in the software of practical flight control systems. In particular, he advocated the use of recovery blocks [17] augmented by error detection using an interval timer, and the provision of a back-up scheduler which would maintain only critical system functions.

Kopetz presented an abstract model of computation and used it to develop probability models of reliability. Real-time systems were handled as a special case and it was shown that various forms of redundancy in such systems lead to substantial increases in reliability.

Substantial hardware fault tolerance was built into the SAFEGUARD system together with limited software fault tolerance. Software error detection took many forms including range and plausibility checks on data. Once detected, software errors were usually handled by suspending the erroneous process if possible, by operator intervention, or by
resetting the entire system. Some critical programs were equipped with specialized error recovery code. The techniques used in SAFEGUARD were rather specific to the needs of that particular project.

2 Characteristics and a Model of Real-time Systems

Consider a set of programs with names P₁,...,Pₙ and a set of distinct integer times T₁,...,Tₘ which represent timing constraints. Let G be a finite, acyclic, directed graph with exactly one node with in-degree zero called the **frame beginning node** and exactly one node with out-degree zero called the **frame ending node**. Each node of G is labelled with either a program name or a timing constraint, and more than one node may be labelled with the same program name. No two nodes are labelled with the same timing constraint. An arc of G can only connect two nodes with different types of label. Thus G is a bipartite graph. The **frame beginning** and **frame ending** nodes are both labelled with timing constraints. If there is a path in G from a node labelled Tᵢ to a node labelled Tⱼ then Tᵢ must be less than Tⱼ. Nodes labelled with program names have in-degree and out-degree one. G is referred to as a **synchronization graph**. Clearly, such graphs are connected.

A synchronization graph represents the process structure of a real-time system. It shows which programs must be executed, and when each must start and finish. A node labelled with a program name indicates the execution of that program and therefore represents a process. Where no ambiguity arises, the phrase 'process F' will
sometimes be used in the remainder of this paper in place of the phrase 'execution of the program named P.' A node labelled with a timing constraint is a point of process synchronization (with other processes and/or the outside world) and the node label specifies the time by which this synchronization must occur. A node labelled with a program name P is connected to two nodes labelled with timing constraints. The graph indicates that the program named P must begin execution at the time specified at the node with an arc to the node labelled P, and must complete execution at or before the time specified at the node with an arc from the node labelled P.

The need to respond to changes in the external environment dictates that real-time systems have an iterative structure. The set of programs constituting the system is executed periodically and the system state is essentially the same at the beginning of each iteration. The time period from one such point to the next is referred to as a frame and corresponds to the time to complete a single iteration of the synchronization graph for a system. The length of a frame in real time (the frame time) is usually quite short (normally less than one second) since frequent outputs from the system are required for smooth operation. This means that any given process will only exist for a short time (at most a frame) which may be contrasted with the characteristics of a general-purpose operating system.

Arbitrarily complex real-time systems can be modelled using such graphs but the resulting graphs may be inordinately large. However, we have found that the simple repetitive structure of practical real-time systems allows their operation to be modelled as cyclic traversals of
acceptably sized graphs, beginning each cycle with the unique node labelled by the earliest timing constraint (the frame beginning node) and ending with the unique node labelled by the latest timing constraint (the frame ending node). Some complex systems demonstrate a natural modularity which can be represented using nested synchronization graphs. A part of the system can be represented as a separate synchronization graph and this used as a node in a graph representing the entire system.

Many real-time systems operate with some programs being executed more frequently than others. Such systems are said to operate with multiple frame rates or multiple iteration rates. This can be incorporated into a synchronization graph by labelling several nodes with the same program name and using a series of timing constraints to show limits on sequential executions.

An example of a synchronization graph is shown in Fig. 1. The program names are P1, P2, ..., P8 and the timing constraints are T1, T2, ..., T7. It is convenient to depict the time labelled nodes of the graph in a linear sequence in order to show the relationship of the timing constraints. Programs P1, P2, P3 and P4 begin to execute concurrently at time T1 and P4 must complete by time T2. Program P5 begins at T2 and both it and P3 must complete by T3. At T3, programs P6, P7 and P8 begin, and they, together with P2, must complete by T4. All programs except P1 then repeat this execution sequence with new timing constraints T5, T6, and T7. Thus this example includes multiple rates of iteration. Time T7 is the final constraint and programs P6, P7 and P8 must complete at this time, as must P1 and P2. The entire execution sequence is then repeated an arbitrary number of times. All
timing constraints are incremented by $T_7 - T_1$ at the end of each iteration. $T_1$ is the frame beginning node, $T_7$ the frame ending node, and $T_7 - T_1$ the frame time.

Although Fig. 1 is a simple example, observations of practical systems suggests that they can be modelled adequately with extremely simple graphs. Fig. 2 is the synchronization graph of a slightly modified form of the software for the Annular Suspension Pointing System (ASPS) [23]. (The ASPS is a computer controlled pointing system which provides extremely accurate pointing of experiments to be carried on the Space Shuttle.)

The concurrency exhibited in practical real-time systems, such as ASPS, is usually limited and predefined. Parallel execution of processes is not uncommon, but the degree of parallelism is restricted and takes the same form during each frame. In principle, processes are being created as required but in a rigid predefined manner. There are none of the complex resource management problems which arise in systems where processes may be created arbitrarily. Interprocess communication is fairly simple and often identical in every frame.

3 Principles of Fault Tolerance

Detailed discussions of the general principles of software fault tolerance may be found elsewhere [1,18]. Only an overview is given here.

In what follows, definitions derived from those given by
Melliar-Smith and Randell [16] for fault, error, and failure are adopted. Specifically:

1. a FAILURE occurs whenever the external behavior of a system does not conform to that prescribed by the system specification,

2. an ERROR (more accurately known as an erroneous state) is a state of the system which, in the absence of any corrective action by the system, could lead to a failure which would not be attributed to any event subsequent to the error,

3. a FAULT is the adjudged cause of an error.

The term 'fault' will be used to refer to any defect within a system which causes the system to enter an erroneous state. For example, a hardware component may malfunction either because of physical degradation or because it was badly designed. Software faults are usually called 'bugs' and are due to mistakes in the design or construction of the software.

Fault tolerance techniques can usually be divided into four constituent phases which, taken together, provide the system with a capability for preventing faults from leading to failures. The four phases are:
1. **ERROR DETECTION.** To tolerate a fault its effects must be detected. Clearly, this can only be achieved by performing checks to determine whether any erroneous situation has arisen.

2. **DAMAGE ASSESSMENT.** Having detected that the system is in error, it will usually be necessary to identify how much of the state of the system has been corrupted. The assessment may be based on assumptions about the flow of information in the system, or on the outcome of further exploratory error checking.

3. **ERROR RECOVERY.** Probably the most important aspect of fault tolerance is the provision of an effective means of transforming an erroneous state of the system into a well defined and error free state. Methods for achieving this transformation can sometimes make good use of the information retained in the erroneous state, but it can be more secure to simply discard the erroneous state and reset the system either to some prior state (a recovery point) or other predesignated state.

4. **CONTINUED SERVICE.** In order to enable the system to continue to provide the service required by its specification, further action may be needed to ensure that the fault whose effects have been obviated does not immediately recur and thus ruin the whole approach. Unless the fault was transient and will not
recur in any case, it must either be rectified or circumvented. These actions are usually referred to as repair and reconfiguration, respectively.

Fault tolerance techniques have received widespread application in hardware [4], but are relatively little used in software. This is largely because the techniques adopted in hardware systems are intended to cope with anticipated faults resulting from physical degradation, and as such are inappropriate for software faults. Faults in software are present from the outset; their characteristics are those of design faults and are necessarily unpredictable. Techniques suitable for providing tolerance to software faults have not been proposed until comparatively recently [11,5]. In order to cope with unanticipated situations, the strategies adopted for the four phases of fault tolerance described above must operate as generally as possible.

Thus, it is advocated that error detection should be achieved by checking that the system is functioning acceptably. It is not suggested that the more conventional approach of checking for specific malfunctions should be discarded, but that negative checks of this type should be supplemented by positive acceptability checks.

An automated exploratory approach to damage assessment would be difficult in an unanticipated error situation. Decisions about the extent of damage are more appropriately based on assumptions of how the system is structured and the apparent severity of the error.

A similar approach to error recovery entails mistrusting any of the state information considered to be damaged and avoiding the use of
recovery techniques which rely on such information. In order to recover from the unpredictable situations which can ensue from design faults, it is necessary to adopt the more drastic alternative of replacing all suspected parts of the system state together with any other parts which must be replaced for consistency. This may involve substantial processing and consequent delay. To minimize this penalty, hardware implemented state restoration mechanisms have been proposed [15].

Finally, in order to achieve continued service after recovery has taken place, some means of preventing a repetition of the original fault must be found. An estimate of the location of a software fault will be needed so that the module containing the fault can be replaced by a stand-by spare. Given the nature of software faults, it is clear that the spare module must be of independent design.

The technique of recovery blocks [17] is based directly on the above principles whereas N-version programming [5] uses an NMV voting check for error detection and replicated states to obviate the need for explicit error recovery; neither technique is directly applicable to other than a single sequential process.

4 Fault Tolerance in Concurrent Systems

While considerable success has been achieved in devising mechanisms to provide fault tolerance in the software of sequential systems, difficulties arise when systems of communicating concurrent processes are considered, particularly if real-time constraints are imposed.
Suggestions in this more difficult area have involved major assumptions about the nature of the concurrency in the system. Randell [17] assumed that processes could be synchronized with respect to the discarding of recovery points, and suggested a technique of "conversations" between processes. Shrivastava [22,21] considered processes which communicate solely in order to share scarce resources, and Russell [19] examined producer/consumer systems. In a slightly more general but much more complex approach, Kim [12] assumed that interprocess communication takes place through monitors and that inputs to a process are considered to be valid by the receiving process.

The basic problem is that if processes can communicate at will, then whenever one process establishes a recovery point (for state restoration purposes) it is advisable for all other processes to do the same. If this is not done, system-wide consistent state restoration may only be possible by rolling back the activity of the system to an arbitrarily earlier point in time. This is the Domino Effect [17] and an example is shown in Fig. 3. Both processes must rollback to point A in order to recover because of the communication which takes place and the way in which recovery points are established. All of the above approaches are aimed at avoiding the heavy overhead incurred with large numbers of recovery points or extensive rollback.

As was discussed earlier, the process structure of real-time systems contains many synchronization points which are usually associated with timing constraints. Synchronization points occur within the process structure where a subset of the processes are synchronized, and at frame boundaries where all of the processes are synchronized. In
fact, much of the synchronization of processes in a real-time system stems from the need to synchronize with the external environment, rather than from any inherent needs of the processes themselves. Thus much more synchronization occurs than would be found in concurrent systems that do not operate in real time. This means that although real-time systems are concurrent, they have a characteristic which is highly desirable if recovery points are to be provided without excessive overhead. The provision of fault tolerance need not involve any changes to the process structure. Such systems are particularly amenable to the application of a modified form of the conversation technique mentioned above.

A set of processes which participate in a conversation may communicate freely among themselves, but with no other processes. Processes may enter the conversation at different times but, on entry, each must establish a recovery point (see Fig. 4). All processes must leave the conversation at the same time since if an error is detected in any participant, every process in the conversation must restore its recovery point and try again. If the conversation structure is used to provide recoverability in a general concurrent system, the necessary state restoration can be automated using a recovery cache [11], which is a form of mechanised incremental checkpoint. It can be used for both sequential and concurrent software. The recovery cache frees the software designer from the need to specify what has to be recorded, saves only a minimum of recovery data, and maintains the illusion that a complete checkpoint has been taken. Although this is conceptually straightforward, if a recovery cache is not supported in the underlying
machine then extensive processing will be necessary to simulate its operation. Presently available computers do not provide a hardware recovery cache although an experimental version has been built for a PDP-11 [15]. Except in particularly simple cases, the overhead of a software recovery cache is prohibitive.

The successful implementation of a fault tolerant real-time system is greatly facilitated by imposing the following restriction on communication between processes of the system. Let timing constraints $T_i$, $T_j$, $T_m$ and $T_n$ and program names $P$ and $Q$ label nodes of a synchronization graph with arcs defined by the ordered pairs $(T_i,P)$, $(P,T_j)$, $(T_m,Q)$, and $(Q,T_n)$. Necessarily, $T_i$ is less than $T_j$ and $T_m$ is less than $T_n$. Communication is only permitted under the following mutually exclusive conditions:

1. from $P$ to $Q$ if $T_j \leq T_m$, (see Fig. 5a)
2. from $Q$ to $P$ if $T_n \leq T_i$, (see Fig. 5b)
3. between $P$ and $Q$ if $T_j = T_n$. (see Fig. 5c)

Note that condition 3 implies $j = n$ also. Since timing constraints are distinct, if $T_j = T_n$ they are the same node of the graph.

The processes $P$ and $Q$ are said to execute sequentially under conditions 1 and 2, and otherwise are said to execute concurrently, in which case $T_j > T_m$ and $T_n > T_i$. Condition 3 (where $T_j = T_n$) is a special case of concurrent execution. Informally stated, two processes execute concurrently if their executions overlap, whereas they execute sequentially if one terminates before the other begins.

Let the timing constraint $T$ and program names $P_1, ..., P_k$ label
distinct nodes of a synchronization graph with arcs defined by the ordered pairs \((P_1, T), \ldots, (P_k, T)\). Processes \(P_1, \ldots, P_k\) execute concurrently but are allowed to communicate because condition 3 is satisfied. If a recovery point is established for each of the processes when they are initiated then the set of processes \(P_1, \ldots, P_k\) are said to be engaged in an exchange. The exchange terminates when all of the processes have terminated (at or before time \(T\)) whereupon all the recovery points can be discarded. Processes in an exchange can be regarded as a single (atomic) process for recovery purposes if all processes in the exchange are restored to their initial state whenever recovery is needed for one process.

An exchange is a very restricted form of conversation. Specifically, it is a conversation which processes enter as soon as they are initiated and exit only when they all terminate. Although conversations can be nested, the definition of an exchange requires two different exchanges to be completely disjoint. These restrictions are imposed deliberately to take advantage of the natural synchronization inherent in real-time systems. By stipulating that communication between concurrent processes can only occur when the processes are members of the same exchange, a straightforward implementation of recovery for a real-time system can be constructed.

The unrestricted framework of conversations could be used as a basis for recovery in real-time systems, but exchanges are proposed here as a simpler alternative imposing stricter control on inter-process communication. Observation of typical real-time systems suggests that these restrictions are easily satisfied and are acceptable in order to
facilitate the provision of recovery. Conversations provide more flexibility, but to take advantage of this involves imposing additional synchronization points on processes and necessitates the implementation of a more complex recovery mechanism.

Practical real-time systems often have characteristics which allow much simpler recovery if exchanges are used instead of conversations. Specifically, full state restoration need not be attempted since, in practice, a great deal is usually known about the system state when processes synchronize. The repetitive nature of a real-time system dictates that its state at a given synchronization point will be very similar on each frame. No data, or very little, is generated which is used or modified from frame to frame. Recovery mechanisms can therefore be based on re-establishing the processes as they normally appear when initiated in a frame and then ensuring that any frame specific data has its correct values. In view of the limited amount of data which is frame specific, the recovery required involves little more than a reset. As such, hardware assistance is not essential. Assuming all code and constants to be in a read only memory, the reset procedure can be simply and adequately handled in software.
5 Error Classification

For the purpose of discussing the recovery mechanisms, errors will be classified according to a set of definitions. This classification is based on the apparent seriousness of the situation arising from a fault. It is not appropriate to classify faults in this way since similar faults occurring at different points in a system can generate erroneous states with different degrees of severity. The definitions are:

1. INTERNAL error – an error that can be adequately handled by the process in which the error is detected.

2. EXTERNAL error – an error that cannot be adequately handled by the process responsible for the system being in error, but whose effects are limited to that process.

3. PERVASIVE error – an error that cannot be adequately handled by the process responsible for the system being in error, and such that other processes generate errors not directly attributable to their own faults.

The incidence of errors will be classified according to the following definitions:

1. PERSISTENT – an error is persistent if the frequency of
occurrence of the associated fault exceeds some predetermined threshold.

2. TRANSIENT - an error is transient if it is not persistent.

As an example, suppose process P is in execution and enters an erroneous state which is detected. The detection may be by the process itself such as an assertion failing, by the hardware such as a division by zero, or by the executive such as P not completing in the required amount of time.

If process P performs a division by zero, and has provision for recovery from this error and this recovery is successful, then the error is considered to be internal. Such an error has no impact on the rest of the system.

Internal errors may be transient or persistent, but this is of no consequence because of their lack of impact. However, a persistent internal error may lead to an external or pervasive error.

If division by zero occurs within P but no error handler is provided, the service provided by P can only be maintained by software which is not part of P. The error is then external, provided that no erroneous information has been propagated from P to other processes. Such an error is persistent if it recurs frequently, and it is likely that more extensive recovery will be required for persistent external errors than for transient ones.

Finally, suppose process P enters an erroneous state by, for example, setting to zero a variable which should always be positive.
If, before this situation is detected, process P uses this variable in communication with other processes and they enter erroneous states as a result, then the error is pervasive.

Given this classification scheme for errors, it is necessary to be able to determine which class an error falls into once it has been detected. This enables appropriate recovery techniques to be invoked reflecting the extent of the damage incurred by the system. In practice, classifying an error can only be attempted since it will be impossible to classify all errors correctly. For example, an error could occur which was in fact pervasive, but if the consequent damage to the other processes was not detected, then this pervasive error would be indistinguishable from an external error. There is nothing that can be done about this problem other than to try to minimize its impact. Some form of recovery will be invoked even when an error is wrongly classified and this may still be sufficient to ensure continued service from the system.

6 Error Detection and Damage Assessment

When software is being executed on a hardware implemented interface, the various checks built into the hardware may be supplemented by assertions in the software. These assertions may be in the processes themselves or in the executive software.

A check which is performed in hardware may reveal an error by detecting an invalid usage of the interface; for example division by
zero, a protection violation, or an attempt to execute an invalid operation code. A software assertion may reveal an error by detecting an illegal use of program data; for example range checking, array bound checking, or checks on invariant relationships between variables. Timing errors are detected when a timing constraint specified in the software is violated. They form an important class of errors detected by supplementary software checking.

If error detection is to be followed by recovery and continued service, there must be time available after detection and before the outputs are actually required. In the particular case of signalling timing errors this means that deadlines must be imposed on the primary process which allow time for recovery and the execution of at least one alternate [3]. This is an important point for system designers to keep in mind.

Errors detected by hardware are usually signalled by the generation of an interrupt, but the signaling of software detected errors can take many forms; for example a flag could be set, a branch instruction executed leading to an error handler, or an interrupt deliberately generated. In the present context, the latter is usually preferable because it allows all errors to be handled uniformly and it probably requires the minimum number of instructions to be executed in the routine which has failed.

It is assumed in the following discussion that whenever an error is detected, either by hardware or software checking, an interrupt is raised and control passes via the hardware interrupt handler to a system error handler.
The error handler must respond to all errors attributed to the software of the system and ideally would itself be provided in hardware. Even where this is not the case, the system organization should be such that the error handler may be envisaged as part of the underlying hardware.

On being invoked in response to the detection of an error, the error handler must make a determination of the extent of the damage to the system state, and then initiate appropriate error recovery measures. In the approach proposed in this paper, damage to the system is implicitly assessed by the error handler classifying each error as being internal, external, or pervasive. For external and pervasive errors, the recovery technique applied is also dependent on whether the error is deemed to be transient or persistent.

In order to classify errors with reasonable accuracy, it will be necessary for the error handler to retain information concerning the error history of processes in the system. No information need be maintained for internal errors since such errors are considered to be completely localized difficulties for which the recovery applied by the process involved is adequate.

Whether an error in a process can be considered an internal error or not will be very system dependent. The error handler makes this determination on the basis of two questions:

1. Is this particular error one for which processes are permitted to attempt local recovery?
2. Does the process in which the error occurred have the means of attempting local recovery for this particular error? For example, did the designer include a suitable exception handler?

If local recovery is permitted and available, then the error handler allows the process to initiate its own recovery capability (such as exception handling or recovery blocks). Only if this recovery apparently succeeds is the error finally classed as internal. Otherwise, the error will be dealt with as an external or pervasive error.

An error can be suspected to be pervasive if multiple non-internal errors occur in a single frame. A persistent external error is suggested if an external error recurs frequently in a particular process. Frequent recurrence of pervasive errors indicates a persistent pervasive error. Quantification of "multiple" and "frequent" in the above yields a well-defined classification algorithm for use by the error handler.

It is suggested here that if an external error has occurred in a frame, then any further occurrence of a non-internal error in that frame should be classified as a pervasive error. It is preferable for the error handler to err, if at all, on the side of caution.

The simple iterative structure of many existing real-time systems suggests that a less rigid approach can be adopted toward determining the persistence of an external error. A straightforward frequency test seems appropriate; for example, an external error in process P could be considered persistent if an external error in P had occurred either in
each of the \( n \) previous frames, or in \( p \) of the \( q \) previous frames (where \( n, p, \) and \( q \) are integers selected by the system designer). A more strict version of the same test might be considered necessary to detect recurring pervasive errors.

The information needed by the error handler in order to classify errors is most simply maintained by recording the recent error history of the processes in a bit matrix \( E \), called the process error matrix, whose row index ranges over the processes of the system and whose column index runs from zero to \( q \), where \( q \) is the value employed in the frequency test above. At the beginning of a frame, all the elements in column zero of \( E \) are set to zero. If a non-internal error occurs in process \( P_i \), then the column zero, row \( i \) element of \( E \) is set to one. At the end of each frame, a one place logical right shift is applied to each row of the matrix. Thus, a value of one in position \( E_{ij} \) indicates the occurrence of a non-internal error in process \( P_i \) in the \( j \)th preceding frame to that which is current. Fig. 6 shows an example of a process error matrix as it might appear at the beginning of a frame for a system of four processes in which process 3 has experienced errors in each of the last three frames. It may be convenient to record the incidence of pervasive errors in a supplementary bit vector which records, for each of the last \( q \) frames, whether a non-internal error occurred in two or more processes.
7 Recovery and Continued Service

7.1 Internal Errors

Recovery from internal errors is only attempted for those errors for which explicit provision has been made in the system design. Techniques for internal recovery by a process include 'ad hoc' repair as a part of a local exception handler [8] such as a PL/I 'ON' unit, or a more general approach such as the systematic state restoration employed by recovery blocks. Continued service is provided in an arbitrary fashion following recovery in an exception handler and more systematically by the alternates in a recovery block under the constraint of having to satisfy the acceptance test. It is inappropriate to discuss the response to internal errors in greater detail because in any given set of circumstances, recovery is highly dependent on the structure of the individual processes involved.

It is important to be aware that the processing of internal errors may be unsuccessful and provision must be made to detect this situation and signal an external error. Detection is trivial when using recovery blocks since it corresponds to exhausting the set of alternates. It is more difficult with exception handling because complete consistency checks of the recovery must be programmed explicitly. Signaling the external error may take any form which is appropriate for the system involved.
7.2 External Errors

The recovery used for external errors will depend upon whether or not the process is engaged in an exchange. For a process which is not, a suitable state can be restored by a simple reset mechanism as discussed in section 4. Under software control, values of input data for the process can be established in preparation for execution of a suitable alternate.

For a process engaged in an exchange, the recovery can be similar but must involve all the processes in the exchange. A suitable initial state must be established for each so that alternates can be executed for each.

It has been suggested [9] that in most real-time systems there are certain processes in which any error leads to a system failure which is critical. Such processes are called critical processes. Other processes are not critical in that if they are in error, the resulting system failure is not critical. An example of the former might be a process responsible for engine throttle settings, while an example of the latter might be a process which provides noncritical information for display. Classification of processes in this way is the responsibility of the system designer but it must be borne in mind that a process may be regarded as noncritical in the presence of transient external errors but may have to be regarded as critical in the presence of persistent external errors. For example, an external error occurring in a process providing information such as fuel level or engine temperature for display, may cause the display to blink or present erroneous information
for an instant if the error is transient. This is probably of no concern. However, a persistent error in that process could lead to a complete loss of the information and may require termination of whatever mission is in progress.

Three general approaches to recovery and continued service are possible following the detection of an external error. They are:

1. No special processing. The error is ignored and the system continues trying to provide service.

2. Provision of behaviour that is acceptable in the short term but is inferior to that intended from the process in which the error is deemed to have occurred.

3. Provision of behaviour equivalent to the intended behaviour of the process in which the error is deemed to have occurred.

Approach 1 could be considered for processes which are classified as noncritical but for no others. It is not recommended even under these circumstances since there is always the danger that an untreated error could have unanticipated side effects.

Approach 2 is essentially the use of recovery blocks as proposed by Hecht [9]. Although it was suggested in the context of timing errors, this approach is equally applicable to other external errors. The occurrence of a fault in a primary process is handled by the execution of an alternate providing degraded service. It is interesting to note
that several simple alternates are possible. In particular, in real-time systems with short frame times it is often acceptable to re-use the outputs of the previous frame as the outputs for the frame in which the error occurred. This is known as the "skip-frame" strategy. Another possibility is some form of extrapolation based on data from several previous frames. For example, an acceptable output might be generated by adding the difference between the outputs of the two previous frames to the output of the previous frame.

Such simple strategies are attractive but great care must be exercised in their use if interprocess communication is taking place. If the communication is between processes which execute sequentially, outputs which are satisfactory for receiving processes must be generated by the alternate. If the communication is between processes which execute concurrently, the process which is in error will have been involved in an exchange and so it is necessary to perform state restoration for all processes in the exchange. For example, suppose a set of processes are designed to produce commands to control surfaces of an aircraft and they communicate in an exchange while performing their calculations. If one of them is in error, the outputs of all of them will have to be mistrusted. Alternates for all of the processes in the exchange will have to be used in such cases. Although the skip-frame strategy seems simple, in practice it may not be because much more processing is needed than simply the preparation of a suitable output for a single process.

Approach 3 is similar to approach 2 but assumes that non-degraded outputs must be generated on every frame regardless of the occurrence of
faults. In practice this approach will be required only rarely in the treatment of transient external errors. Most real-time systems seem able to operate acceptably despite momentary degradation of service and, if an external error is truly transient, approach 2 will often be appropriate. If an external error is persistent, repeated use of approach 2 is very likely to result in system failure eventually. For example, repeated use of the skip-frame strategy amounts to the system repeatedly ignoring changes in the external environment. The primary intent of most real-time systems is prompt response to changes in the external environment.

Hecht [9] has proposed the design of a real-time executive which will remove a defective process from the system and replace it by a new version. Using the model and error classification scheme proposed here, this amounts to responding to a persistent external error by replacing the relevant process with a substitute. This substitute should be completely equivalent in its interfaces to the rest of the system but constructed differently so that, hopefully, it will not become erroneous under the circumstances which caused the original process to become erroneous. It is worth noting that even when a defective process has been removed, it can still serve as a stand-by spare in case the substituted process is found to be defective under different circumstances at a later time.

Thus, provision of continued service depends on whether the external error is transient or persistent. Both types can occur and so provision must be made for both. This suggests that every primary program should be supplemented by at least one alternate program capable
of providing degraded service to cope with transient external errors and another version of the primary program to cope with persistent external errors.

7.3 Pervasive Errors

Pervasive errors are the most serious of the error classes. The notion of critical and noncritical processes does not apply in the presence of a pervasive error. The fact that the error is pervasive means that, in the absence of fault tolerance, failure of critical processes is very likely even if the process error matrix $E$ only indicates the occurrence of errors in supposedly noncritical processes. So much damage has probably been done that critical processes will almost certainly enter erroneous states.

Strategies are limited by the gravity of the situation. The error will be classified initially as transient and the only practical approach to continued service is to use the simple skip-frame strategy discussed above. The time required to attempt the execution of more elaborate alternates for many processes is almost certainly unacceptable. If the error is indeed transient then the skip-frame strategy is probably adequate anyway.

If the error turns out to be persistent and pervasive then it is extremely unlikely that the system will be able to provide any acceptable service. Treatment of the error during its initial transient classification will have attempted to ensure that acceptable service was
maintained but such treatment cannot continue. The only viable automatic treatment for persistent pervasive errors is complete replacement of the software. If provisions for recovery and continued service have been made for external errors, there will be a second version of each process available and the replacement of each process by the second version amounts to total software replacement. Once again, recovery can be handled by a simple reset.

8 Conclusion

A classification scheme for errors and a technique for the provision of software fault tolerance in real-time systems have been presented. The technique is considered to be evidently practicable because of its relatively simple approach. It has been argued that many real-time systems have characteristics which make them particularly amenable to the inclusion of fault tolerance using this technique.

In summary, the technique requires that the process structure of a system be represented by a synchronization graph which is used by an executive as a specification of the relative times at which programs are to be executed and the way in which they will communicate during execution. Communication between concurrent processes is severely limited and may only take place between processes engaged in an exchange. A history of error occurrences is maintained by an error handler. When an error is detected, the error handler classifies it using the error history information and then initiates appropriate
recovery action.

There are costs associated with the provision of fault tolerance; both in the implementation and operation of a system. Operational overhead is less important because it can be traded for an increase in hardware resources. However, the additional costs in design and construction of the software may be substantial. If two versions of a primary process are to be provided they must both receive equal care and attention in their preparation. It might be expected that this would more than double the total cost of the software but Gilb [6,7] has argued that producing two versions of a software module should only cost about 10% more than a single version.

It must be remembered that in such critical systems as commercial air transports, the software cost is not a substantial portion of the total development cost. Copies of the software for additional aircraft cost nothing and so, for an entire fleet, the cost of producing high quality fault tolerant software may be insignificant compared to the total cost of producing the aircraft. Irrespective of the cost, in many cases the need for the utmost reliability dictates the need for fault tolerant systems.

9 Acknowledgement

It is a pleasure to acknowledge the very helpful comments on an earlier version of this paper which the authors received from B. Randell of the University of Newcastle upon Tyne and R. H. Campbell of the University of Illinois.
References


Fig. 1. An Example of a Synchronization Graph.
Fig. 2. Synchronization Graph of the ASPS System.
fig. 3. The Domino Effect.

fig. 4. A Conversation.
Fig 5. Communications Restrictions.

Fig 6. Error Matrix Example.