Process Survivability in a Distributed Computer Control System.

Peter Trueman B.Sc. M.Sc.

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

University of Keele

1985

Abstract

Possibly the greatest advantage that a distributed computer control system has over a centralised control system is that the failure of one or more of its constituent computers does not prevent the other computers from operating normally. Unfortunately, the loss of the executive and application software hosted by a failed computer will prevent the surviving part of the control system from fulfilling its role. Whereas it is possible to design the executive software so that the loss of one of its constituent kernels will not prevent the others from functioning normally, it is not possible to do this for the application software.

Process survivability was conceived as a way of preventing application processes from being lost as a result of a computer failure. Process survivability enhances the high availability of a distributed computer control system's hardware by making its application software invulnerable to computer failure. Process survivability is performed in a way that is transparent to the application programmer.

In this thesis we first describe the distributed computer control system called PROSUR (PROcess SURvivability) which we designed as an environment in which to develop process survivability.

The major part of this thesis is concerned with the design and development of process survivability for PROSUR. In particular, we describe how redundant inactive copies of all of the application processes are incorporated into the application software and how the processes are recovered to a consistent state after a computer failure.

As well as showing that process survivability is practicable, we also investigate its practicality. A simulation study of a distributed computer control system incorporating process survivability has been performed to gain an insight into the effects that process survivability

might have on a control system's performance. The results of this simulation are presented and a number of interesting conclusions are drawn.

To J

Acknowledgements

I would like to thank Dr.K.H.Bennett, my supervisor, for the help and guidance that he has given me throughout my research. I would also like to thank Professor C.M.Reeves for his encouragement.

The financial support of the Science and Engineering Council of Great Britain is gratefully acknowledged.

Finally, especial thanks are given to my parents and to Julia for the support, encouragement and assistance that they have always given me.

Contents

1. Introduction

1.0 Introduction

Once a computer has been introduced into an organisation it quickly becomes indispensable and the parent organisation has to rely on the computer's continued correct operation. As familiarity with the computer's powers increases so does the work load that is placed on it. This in turn means that the organisation's ability to operate normally without the computer decreases and the reliance placed on the computer's continued operation increases.

When such reliance is placed on a computer any interruption to the service that it provides (down-time) is much more than an inconvenience. In an on-line data processing system down-time costs money, and in a process control system down-time may not only be expensive, it may also result in a situation arising where lives are endangered. Protracted down-time can result in the financial collapse of a commercial organisation: a recent survey in the United States of America found that "80% of companies would fail to survive a large scale computer disaster" (reported in Bremen 1983).

A computer is said to have failed when it ceases to meet the specifications that define its behaviour. A computer's reliability is normally measured by the probability that it will not fail during a specified time period. The higher the probability and the longer the time period, the more reliable the computer is.

Obviously there is potentially a large market for reliable computers. Unfortunately computers are not inherently reliable as they are made up of unreliable components which are combined in a complex manner. Component reliability is continually being improved but it is very unlikely that it will ever reach 100%, and even if it did, the extra complexity

resulting from combining the components together would bring the computer's reliability down to well below 100%. The alternative to producing a computer that never goes wrong is to produce a computer that can continue to operate normally despite component failures.

An early but still flourishing market for highly reliable computers is the aerospace industry. Here computers have been used to support the manned space flights and to pilot the deep space probes. More recently, on-board computers are assisting pilots to fly marginally stable aircraft by making the myriad minor alterations to the aircraft's control surfaces that are needed to keep the plane in the air and which are beyond the pilot's capabilities to perform. Such applications require a computer whose probability of failing during the aircraft or spaceship's mission time is so small that it can be ignored. (In an aircraft, computer failure must be as unlikely as the wings falling off.) In order to obtain these high reliabilities computers have been developed that are fault tolerant, that is they are able to continue executing their programs normally and without interruption despite component failures.

Fault tolerant aerospace computers are based on the incorporation of protective redundancy into the computer's design. Protective redundancy is comprised of the extra hardware components, processing power and programs that are not needed to execute the computer's programs, but are there to provide fault tolerance either by masking component faults or by replacing components when they are detected to be faulty (Carter and Bouricius 1971). These computers do indeed achieve the high level of reliability demanded by the aerospace industry. For example, the Fault Tolerant Spacebourne Computer has a 95% probability of remaining operational for five years (O'Brien 1976).

Such computers can also withstand limited physical damage, but as they are physically centralised it is likely that any major physical damage, such as that resulting from a fire, would completely destroy them. In fairness, it should be noted that these computers are not designed to cope with such contingencies, as such a situation would most likely doom the aircraft or spaceship anyway.

In the majority of applications the threat of physical damage must be taken into consideration. Numerous examples of organisations being left computer-less after the complete destruction of their computers abound - an interesting selection of such stories can be found in Berthelsen 1983, Lawrence 1983 and Wong 1983. These risks can be reduced by careful management and by codes of practice, but they can never be removed. For example, a recent UK survey showed that 47% of fires affecting computer installations started outside of that installation (Wong 1983). In fact, in certain applications it may be impossible not to place the computer in a hostile environment where such damage is likely - a computer in a warship is one such example.

As the physical threat cannot be removed, the computer system must be built to tolerate physical damage. This can be done in the same way that component failure is tolerated except this time the unit of hardware redundancy within the system must be the computer itself. By basing the system on more than one computer and by dispersing these computers, the system cannot be totally destroyed by a single disastrous event at one site.

The invention of the local area network, the falling cost of processing power, and a growing demand for a system whose characteristics include tolerance to computer failure have led to the development of distributed computer systems. The title 'distributed computer system' has many meanings, but in this thesis we use it to describe a collection of

computers that are physically interconnected by a local area network, and which are logically interconnected to form a single unified system.

When one of the constituent computers of a distributed computer system fails it does not prevent the remaining computers from operating normally. The overall computing power has been reduced but the majority of the power is still available. Furthermore, the physical dispersal that can be achieved using a local area network effectively removes the chance of all of the computers failing at the same time.

The crash of a computer results in the loss of the executive and application software that was running on that computer. If this software provided a service that is vital to the continued running of the surviving computers' software, then although the majority of the distributed computer system is intact it will no longer be able to fulfil its role. The distributed computer system as a whole is still vulnerable to computer failure.

Distributed databases have been developed as a partial solution to this problem, as by replicating the data they ensure that a computer failure does not prevent access to vital data. However, all the other software resources provided by that computer would be lost and these may be equally vital.

The aim of the research described in this thesis is to investigate how the vulnerability of a distributed computer system's software can be removed so that the full potential of its distributed hardware can be exploited. The result of this research is process survivability.

Distributed computer systems can be used for real-time work, ranging from process control in a chemical works to on-line transaction processing in a bank. Distributed computer systems used for such applications are called distributed computer control systems. Process

survivability has been developed for inclusion in a distributed computer control system because the cost of the control system failing, whether measured in money or lives, is greater than in any other application.

1.1 Distributed computer control systems

As we shall show in Chapter 2, a distributed computer control system consists of three levels: hardware, distributed kernel and application.

The hardware consists of a number of computers linked together by a local area network (Clark et al. 1978, Gee 1983). Devices are attached to the computers.

Each computer has a kernel. The kernels cooperate with each other to form the distributed kernel level. The distributed kernel level provides the logical unity that is a characteristic of distributed computer systems. This unity is achieved by hiding the distributed nature of the hardware from the application level, thereby providing it with a single-computer environment in which to operate.

The application level consists of a number of application processes that cooperate with each other to implement the control system's role. Cooperation is by an interprocess communication mechanism which is provided by the distributed kernel level.

When a computer crashes it ceases to work and the kernel and the application processes that are running on it are lost. If the control system is to continue then the distributed kernel level must be able to operate without the lost kernel, and the application level must be able to operate without the lost application processes.

As we shall explain in Chapters 2 and 4, kernels generally operate autonomously and only cooperate with each other to support interprocess communication between application processes on different computers. Because of this autonomy the loss of a kernel will not prevent the others from working normally.

Application processes are not normally autonomous. Some application processes provide unimportant services such as error logging, and the absence of these would not prevent the rest of the control system from operating. Other application processes however provide services that are vital, and without them the surviving application processes will not be able to operate properly.

Process survivability prevents an application process from being lost in a crash. By applying process survivability to every application process the application level is made crash tolerant. The application level will be unaffected by a computer crash.

Unfortunately, even with process survivability the control system as a whole is still not crash tolerant, as it is vulnerable to device loss as well. When a computer crashes the devices attached to it can no longer be used, and without these the control system's ability to fulfil its role is reduced. To remove this vulnerability the devices themselves would have to be replicated. Although this problem is outside of the scope of process survivability we return to it in several of the later chapters.

The control system is also vulnerable to the failure of its network. If the network fails the control system will be partitioned into small groups of computers. Again this vulnerability can be removed by replication.

1.2 Process survivability

Process survivability ensures that application processes survive the crash of their host computer by causing those processes to migrate to other computers. Process survivability effectively reconfigures the application level to avoid the crashed computer. This reconfiguration is transparent to the application processes.

Process survivability is achieved by incorporating non-running copies of all of the application processes into the application level. Non-running backup copies of each application process are placed on different computers. At intervals the backup copies are updated so that they form exact copies of their application process.

When an application process is lost in a crash, one of its backup copies is activated to replace it. In effect the application process has side-stepped the disaster, having been moved from the crashed computer to one of the survivors. The remaining backups ensure the survivability of the new generation of application process. By having multiple copies of each process the system can withstand the simultaneous destruction of a number of computers.

By applying process survivability to every application process we ensure that in the event of a computer crash no application process is lost. Process survivability makes the application level crash tolerant.

Process survivability has been designed to have the following advantageous features:

a) Process survivability is transparent to the application programmer: process survivability does not involve any code in the application process, and so it will not fail due to programmer error.

b) As long as the number of computers that have failed is within the level of process replication, crash tolerance can be maintained.

c) Multiple backups of each process ensure that crash tolerance is not affected by simultaneous computer failures.

d) Crash tolerance is not affected by what a process is doing when its host computer crashes. Process survivability coverage is 100% (assuming that a backup copy of that process still exists).

e) Process survivability is automatic. There is no operator involvement other than specifying the level of redundancy required and the position of the backups.

This thesis describes how process survivability can be implemented for a 'paper' distributed computer control system called PROSUR (PROcess SURvivability). Part of the preliminary work for the research described involved designing PROSUR so as to create an environment in which process survivability could be developed. PROSUR is called a paper system because its characteristics have been specified on paper but it has not been implemented.

1.3 The cost of process survivability

All fault tolerance is based on some form of protective redundancy - hardware, software or time (program execution). Process survivability is no exception.

In a number of distributed computer control system applications, maintaining the control system's response time to external events is important. After a computer crash backups will be activated to replace the lost application processes and the surviving computers will be running more application processes than before. If the control system's response time

is not to fall below acceptable levels, the computers must be configured with sufficient spare processing power to absorb the extra load. Once this spare power has been exhausted, further computer losses, although not causing the control system to fail, will cause the control system to start missing deadlines.

To be able to withstand n computer failures without its response time degrading, a distributed computer control system must contain time redundancy equivalent to the power of n computers. Thus the control system has n computers that are surplus to its normal requirements and these are the major cost of process survivability. (There are no designated standby computers in the system; the active application processes are partitioned between all of the computers.)

This expense can be spared if a degraded response time is acceptable after a computer failure. The distributed computer system can be configured without any computer redundancy, just sufficient computers to fulfil its primary role and no more. (The absence of computer redundancy does not mean that there is only one computer.) In the event of a computer failure the control system's response time would become slower.

Further protective redundancy in the form of extra code, processing, and communication network activity is needed to implement process survivability. All of this will either reduce the amount of profitable work that can be performed by the system, or will reduce the system's response time. A simulation study of a distributed computer control system incorporating process survivability has been performed in order to ascertain what these overheads are likely to be, and to what extent they might effect the control system's ability to perform useful work. The results of the simulation and the conclusions drawn from them are presented in a later chapter.

The inclusion of protective redundancy in any form increases the price of a system without increasing its through-put. Normally, a fault tolerant system will only be adopted when the cost of a system failure is greater than the cost of preventing that failure.

1.4 Possible applications

We believe that a distributed computer control system incorporating process survivability will be of greatest value in process control applications. Such applications require a crash tolerant service with a guaranteed response time which will not degrade even in the event of multiple failures. The cost of providing such a service should be justifiable on both financial and safety grounds.

A particularly extreme process control application is that of a warship command and control system as the environment it operates in is hostile and the cost of a system failure is high. "Military command and control...systems must be able to operate fast, move fast or hide, and function in the presence of physical (as well as other) counter measures" (Licklider and Vezza 1978).

A naval command and control system controls the ship's weapons, sensors and displays in order to assist the command team in their decision making. The command and control system monitors the ship's environment using various sensors such as radar and sonar, and presents this information via displays to the command team. The decisions taken by the command team are in turn executed by the command and control system.

A warship and its crew are dependent on the warship's command and control system. If it fails in action the warship will quickly become untenable. Current command and control systems are based on a single computer thus making the ship vulnerable to a single hardware fault, and to light action damage that would not otherwise have affected the ship.

By basing the command and control system around a distributed computer control system that incorporates process survivability, a lot of this vulnerability would be removed. Configuring the distributed computer control system to include a high level of computer (and time) redundancy should enable the command and control system to withstand action damage for as long as the ship remains a viable weapons platform. Dispersing the computers around the ship will reduce the chances of action damage destroying all of the computers in one go. If multiple crashes occur, process survivability's use of multiple backups for every application process ensures that the application processes survive.

Ideally the cost of the computer redundancy would be easily justified by the need of a warship to remain operational. In practice however, the level of redundancy would be based on the expected threat, the cost of providing that redundancy and the amount of space available to put it in.

The value of basing a command and control system on a multi-computer architecture has already been recognised. The Canadian navy has adopted such a system for use in their warships as a way of limiting the physical effects of action damage, but they do not attempt to achieve anything like process survivability (Carruthers 1979). In Britain, the Admiralty Surface Weapons Establishment are currently developing a distributed computer control system for use in Royal Navy warships (Rowland 1982). This system does incorporate crash tolerant application processes, and it is the subject of later discussion in Chapters 2 and 4.

Any commercial institution that supports an on-line transaction processing system for its own use, or for the use of its customers, would benefit by adopting a crash tolerant computer system, as even a small interruption in the computer's service would lead to a loss of money and loss of customer goodwill. One of the earliest examples of an on-line

database, the SABRE airline reservation system (Plugge and Perry 1961), was based on a fault tolerant computer system.

The threat of a major disaster completely destroying a computer is now being taken very seriously, and has led to a number of 'hot standby' sites (Harrington 1983, Berthelsen 1983) being set up as commercial ventures. Hot standby sites are computer suites that can be taken over by a client in the event of a disaster destroying his own computer system.

Basing an on-line transaction processing system on a distributed computer control system that incorporates process survivability would remove the threat of loss of service due to hardware faults or disasters. The amount of protective redundancy that needs to be incorporated in such a system is much less than in a command and control system, as the chance of more than one failure at a time is remote. Process survivability's ability to provide a crash tolerant service without any computer redundancy, albeit at the risk of a degraded response time in the event of a failure, makes it attractive for commercial applications.

This 'no extra cost' configuration also means that a crash tolerant service could be provided where the extra cost could not normally be justified, for example as a University time-sharing system.

1.5 Summary of contents of thesis

Process survivability is based on the use of standby redundancy within the application level of a distributed computer control system. The first three chapters review the background to process survivability.

Chapter 2 describes the organisation of a number of existing distributed computer control systems and describes the three-level model outlined in Section 1.1. Chapter 3 reviews the theory of fault tolerance and gives two extended examples of the use of standby redundancy to achieve

fault tolerance.

Chapter 4 describes the two ways in which redundant copies of application processes can be incorporated into the application level. A major example of each is given. The rest of the thesis describes process survivability.

Chapter 5 reintroduces process survivability. Its aims, characteristics, and relationship to previous work are described. The succeeding chapters then describe how process survivability can be added to a specific example of a distributed computer control system - PROSUR.

Chapter 6 describes PROSUR, the model distributed computer control system that has been used as an environment in which to develop process survivability. PROSUR complies with the three-level model presented in Chapter 2, except that it has a fourth level interposed between the application level and the distributed kernel level. This level, the process survivability level, supports the process redundancy incorporated in the application level. The main emphasis in this chapter is on how redundant copies of application processes are organised, maintained and used.

An application process's backups are updated at intervals to become exact copies of that application process. So, when a backup is activated it will be out of date, and any outstanding messages that have been sent to its previous generation will have been lost. The combination of these two effects means that the restarted application process's state will be inconsistent with the state of the rest of the application level. Chapter 7 discusses these inconsistencies in detail.

Part of the process survivability level's role is to remove these inconsistencies. Chapter 8 introduces the way in which consistency is restored. Finally, Chapter 9 presents full details of how the process

survivability level would be implemented.

We have developed a way of implementing process survivability. But is it practical, or will the overheads caused by its inclusion reduce the level of work being done to a prohibitively low level? A simulation study of PROSUR has been performed in order to examine this question. The simulation's aims, implementation and results are described in Chapter 10.

Finally in the Conclusion (Chapter 11) we review the work presented in this thesis, and describe the possibilities for the further development and exploitation of process survivability.

2. Distributed Computer Control System Organisation

2.0 Introduction

A real-time computer system monitors the state of its environment and reacts to events that occur within its environment. A real-time system can be divided into the controlled system and the control system.

The controlled system comprises those devices that provide an interface between the control system and its environment. The actual composition of the controlled system will depend on the function of the real-time system. For example, in an on-line transaction processing system it may consist of disks, VDUs and line printers, and in an industrial process control system it may consist of thermometers, valves and relays.

The control system is the regulating element in a feedback loop. It consists of the computer programs that perform the regulating and the computer hardware that those programs run on.

A distributed computer control system is a control system that is based on a number of computers linked together by a communication network. The control software is dispersed between the computers with the individual software components cooperating with each other to fulfil the system's overall role.

In this chapter we describe the software and hardware structure of distributed computer control systems. Unfortunately there are as many ways of organising a distributed computer control system as there are distributed computer control systems. So we first review a selection of existing and proposed systems, and then we present a generalised description of a distributed computer control system. First however, we review the potential advantages that can be attributed to a distributed system.

2.1 The potential advantages

Being based on a number of interconnected computers, a distributed computer control system's hardware has a number of extremely advantageous characteristics:

1) The computers can be placed close to (or even within) the devices they use, thus reducing the control system's response time to external events. This will be particularly useful in an industrial complex where the controlled system is perforce widely dispersed.

2) Processing can be performed in parallel, again reducing response time. Large tasks can be divided into smaller concurrently running tasks, and events that occur in parallel in the system's environment can be handled in parallel. The latter ability may be crucial in an emergency when alarms are generated by a number of sources.

3) Such an architecture makes it possible to exploit the 'enough' principle (Brenner et al. 1980). The enough principle states that by reducing hardware resource restrictions it is possible to produce simpler software. Hardware costs are continually falling, especially where established technology is concerned, but software costs continue to rise, and so it is now economical to use hardware to reduce software complexity and hence reduce the overall cost of a control system.

4) The architecture is very modular. The computers communicate via a common medium - the communication network. As long as a computer can use the network in the standard way, it can communicate with the other computers. This means that the type of computers used can be tailored exactly to the requirements of the system, and as these requirements change, the hardware can be easily extended by the addition of other computers. Thus the system's processing requirements are met very economically.

5) The final and possibly the most important advantage is the high availability of the system. When a computer suffers a hardware failure it cannot prevent any of the other computers from running. The overall system has suffered a reduction in its processing power, but the majority of the power is still available. It is this characteristic that is exploited by process survivability. (The system is however reliant on the continued operation of the network.)

The first three of these advantages can be easily exploited by the designer of a distributed computer control system. However if the control system is to benefit from the last two of the advantages, then its software must be designed so as to exploit them to their full potential.

2.2 Existing distributed computer control systems

2.2.0 Introduction

In this section we briefly review the hardware and software layout of a number of distributed computer control systems. From these descriptions it will be seen that although details vary between systems their overall structure is very similar.

In the following descriptions we have where possible standardised the terminology used rather than use the authors' original terminology.

2.2.1 CONIC

CONIC (Kramer et al. 1982, Lister et al. 1980) is designed for use as an industrial process control system. Unfortunately, neither of these papers gives an explanation of what the acronym CONIC stands for.

The envisaged hardware structure is that of a group of computers interconnected by a mesh of subnetworks linked together by gateways (Sloman 1982).

Software consists of a number of concurrently running processes. Processes that cooperate to perform a particular service are grouped into a structure called a module. The internal structure of a module is hidden from other modules. A module must be located on a single computer, but there may be more than one module on a computer.

Interprocess and intermodule communication is provided by asynchronous message passing and by a request-reply message pair similar in operation to a procedure call. Furthermore, interprocess communication within a module can be performed by shared data.

In CONIC the software is divided into a hierarchy of distinct levels: kernel, communication system, operating system, application software and management software. Other than the kernel level, all of these levels are implemented using modules, processes and message passing.

The kernels support multiprocessing and intra-computer message passing, and they provide the routines needed by the local operating system. Inter-computer message passing is performed by a communication module on each computer. The distinction between inter-computer and intra-computer communication is hidden from the higher levels.

The operating systems on each computer provide the services needed to configure and reconfigure the application software, and management software. The management software monitors the system's activities and controls the system's installation.

2.2.2 Distributed Sensor Network (DSN)

DSN (Rashid 1980) is a fault tolerant system developed at Carnegie-Mellon University. It is based on Three Rivers Corporation PERQs connected together by a network similar to Ethernet (Metcalfe and Boggs 1976).

The basic software unit is the process which performs a computation and/or manages resources. Each computer has a kernel and a set of processes. It is assumed that the processes will be organised into a hierarchy of levels.

Application processes may be written in a number of languages, each of which will have its own interprocess communication mechanism. All of these mechanisms are implemented using asynchronous message passing.

The kernel provides virtual memory management, multiprocessing, local interprocess communications, routines to control the devices, and routines to create and delete processes. Inter-computer communication is provided by a network process resident on each computer. This partitioning of labour is invisible to the application processes.

2.2.3 A Signal Processing System

This system (Stepczyk 1978) is a dedicated system designed to perform signal processing.

The hardware consists of three PDP-11's linked together by unidirectional bus links to form a ring. All of the software is written in Concurrent Pascal (Brinch Hansen 1977). Each computer runs a virtual machine environment for supporting Concurrent Pascal, and a number of processes. Interprocess communication is by synchronous message passing and by shared data in the form of monitors.

Message passing is implemented by two monitors on each computer. One performs intra-computer message passing and the other inter-computer message passing, with the partitioning being hidden from the user processes by the class that provides the message passing routines.

Shared data is stored in monitors. Only local processes can use a monitor's procedures. Every process that requires remote access to a monitor has a 'ghost' process local to that monitor. To use a monitor a remote process sends its ghost a message which the ghost translates into a call to the monitor. The results of the monitor call are then returned by the ghost to the remote process in a message. In a later version remote access to monitors will be allowed.

2.2.4 Action Data Network (ADNET)

ADNET (Moulding 1980a and Moulding 1980b) is an experimental warship command and control system developed by the Admiralty Surface Weapons Establishment (ASWE).

ADNET is based on a number of Ferranti Argus minicomputers linked together by an ASWE Serial Highway (Ministry of Defence 1981, Hill and Stainsby 1980). The software is implemented in Coral 66, supported by Mascot.

The application software consists of a set of processes partitioned between the computers. The processes cooperate with each other using asynchronous message passing. The application processes on each computer are supported by a local copy of the Mascot Operating System (Miles 1980) and a communication package.

The Mascot Operating System supports only a single-computer Mascot environment. All interprocess communications, whether inter-computer or intra-computer, are handled by the communication package on each computer.

Further details of ADNET are given in Chapter 4, Section 4.2.2.

2.2.5 Distributed Processes

'Distributed Processes' is a programming language concept proposed by Brinch Hansen (1978) for programming distributed real-time systems. Although he concentrates on language features there are sufficient details about the control system's configuration and implementation to justify its inclusion here.

Distributed Processes is designed for programming a network of microcomputers linked together by a communication network. The number of processes in the system is fixed and to ensure adequate response time there is only one application process per computer. Interprocess communication is by procedure calls.

Each computer runs a runtime environment for supporting distributed processes, a single application process and a number of 'ghost' processes. The ghosts are part of the remote procedure call implementation, and are similar to the ghost processes used in the Signal Processing System described above.

A process's procedures are used by only a few processes. Each of these processes is represented by a ghost process resident on the server process's computer. When a user process executes a remote procedure call the parameters are passed to its ghost. The ghost performs the procedure call and then returns any results. Thus the remote procedure call is actually based on message passing at a lower level. A similar scheme is also used in Unix United (Brownbridge et al. 1982), although here the number of users is unknown and so the ghosts are dynamically created as needed.

Multiprogramming the application process and the ghosts is simple. There is no preemption and the processes simply execute until they perform an explicit wait on some condition.

2.3 A generalised distributed computer control system organisation

2.3.0 Introduction

Although the previous section is not an exhaustive survey of distributed computer control systems, it is sufficient to show that although details vary between systems their basic organisation is similar. In this section we present a generalised distributed computer control system model which has each of the above examples as special cases.

Three main levels can be identified: application level, distributed kernel level, and hardware level. We will now describe each of these in turn.

2.3.1 The application level

The application level is divided into a number of application processes, which cooperate with each other in order to fulfil the control system's role. An application process is a named and executable instance of a program. The application processes are dispersed between the computers. The location of some application processes will be determined by the location of particular hardware resources, for example an application process that controls a line printer must be hosted by a computer that is connected to a line printer.

The application level may be structured internally by organising the application processes into a hierarchy of levels, and/or by grouping the application processes into modules.

Often a group of application processes will cooperate with each other in order to provide a particular service. In CONIC and in the distributed programming language proposed by Liskov (1979), application processes are grouped into modules, and the module is used as the standard unit of application software. In both examples, all of the module's constituent application processes must be on the same computer.

The application processes (and the modules, if they are used) may be logically organised into a hierarchy of levels. The number of levels in a real-time system is much fewer than in most general purpose computer systems, often as few as two (Boebert et al. 1978). Prince and Sloman (1981) identify four hierarchical levels in an industrial process control system: direct control of devices, subsystem control, site-wide control, and management control. In other distributed computer control systems, such as the signal processing example above, there is no obvious layering at all.

In traditional computer systems there is an operating system level between the kernel level and the application level, which provides access to system resources. In a real-time system the distinction between system and application resources is negligible, and so the operating system and application software are indistinguishable.

Application processes will be written in a high level language such as BCPL (Richards and Whitby-Strevens 1980), C (Kernighan and Ritchie 1978) or Pascal (Jensen and Wirth 1978). The language will incorporate commands for interprocess communication and for interfacing the application process with the runtime clock. The latter commands include finding the current time, and suspending the caller for a period of time or until a certain time.

Interprocess communication can be performed in a number of ways: asynchronous message passing, synchronous message passing, asynchronous procedure calling and synchronous procedure calling. Numerous attempts have been made to show which of these is the best, and a selection of these arguments can be found in Lauer and Needham 1979, Staunstrup 1982 and Stroustrup 1982. The common result of these comparisons is that there is no difference and that a choice should be made on the efficiency of the primitive operations needed to support the mechanisms.

A more relevant survey into the interprocess communication needs of a distributed computer control system was performed as part of the CONIC project by Kramer et al. (1981). They identified the various interactions that occur between application processes, and concluded that these could be accomplished by using synchronous procedure calling for command-reply and query-status transactions, and by using asynchronous message passing for sending alarms, status messages and delayed responses to procedure calls (thus emulating asynchronous procedure calls). In CONIC itself a request-reply message passing construct is used instead of a synchronous procedure call.

Despite Kramer et al.'s considered arguments there is no apparent consensus of opinion between existing systems. In the examples given in Section 2.2 all of these mechanisms are represented.

A fifth way of performing interprocess communication is by using shared data. This is normally rejected as it would require some form of global virtual address space. However, it is used in CONIC for intra-module communication, but as a module's processes are all located on the same machine, this problem is avoided.

For simplicity and precision in the following discussion, we assume that interprocess communication is by asynchronous message passing. All of the arguments presented below apply equally to the other interprocess communication mechanisms. Asynchronous message passing was chosen because the two major examples in Chapter 4 use it, and so does PROSUR, the distributed computer control system incorporating process survivability. First we describe asynchronous message passing.

For two application processes (the sender and receiver) to communicate by asynchronous message passing they must be connected by a channel. A channel is formed by joining together the sender's output channel to the receiver's input channel. The sender sends messages to its output channel and the receiver receives these messages from its input channel. Messages that have been sent but which have not been received are buffered within the channel. The number of messages that can be buffered within a channel is limited, and if the sender tries to send a message to a full channel then it is suspended until space is available. Similarly, if the receiver tries to receive a message from an empty channel then it will be suspended until a message arrives.

Following Kramer et al. (1981) an application process's response time is defined to be the time taken by an application process to recognise that a message is waiting to be received plus the time it takes to service that message. For a lot of applications, maintaining the application processes' response time within some critical bounds is an important consideration.

From the time at which an application process issues a command to send or receive a message until that command finishes, the application process is suspended. The application process may be delayed by the channel's flow control, or if it is a sender and the receiver is on a different computer it will be delayed by the time taken to transfer the

message between computers. If an application process is to maintain its response it is important that the time spent sending or receiving a message has an upper limit. To achieve this the send and the receive commands have a timeout period. If the command takes longer than the timeout period to complete, then the command is aborted.

An application process will normally have more than one input channel. The pattern of message arrival on these channels cannot normally be predetermined. Messages are equally likely to arrive on any input channel and so an application process must be able to wait for a message from any of its input channels. Kramer et al. (1981) suggest the use of guarded commands (Hoare 1978) which will allow receives to be issued on a number of input channels at the same time, and as soon as one of the receives terminates, all of the others are aborted.

Prince and Sloman (1981) have identified the following three patterns of information flow between application processes:

1) One-to-one: a message is sent by one application process to another.

2) One-to-many: a message is broadcast by one application process to several others, each of which will receive a copy. This might be used for raising an alarm or for requesting a service from a number of identical servers.

3) Many-to-one: an application process receives messages from more than one application process. A process that provides a service may receive requests from a number of users.

In CONIC for example, these information flow patterns can be explicitly incorporated into the application level's configuration. To create a one-to-many connection a single output channel is linked to several input channels, and then when a message is sent, a copy of that

message is placed in each of the input channels. Similarly, a many-to-one connection is created by connecting a number of output channels to a single input channel. Messages sent down any of the output channels will be buffered in the input channel from where they can be received.

Alternatively, these interconnections could be created by using one-to-one connections only. This would not be as elegant or as efficient. For example, in the Distributed Processes System, if a process has to communicate with several other processes, then it has to call procedures in each of the recipients in turn.

For a message to be delivered, the physical address of the recipient's input channel must be known. Using physical addresses within the application processes is impractical as the final address of the recipient may not be known when the processes are written. Instead the processes are identified by logical names.

Before a message can be sent the receiver's logical name must be translated into its physical address. This translation can be made at one of many points: compilation time, system creation time or every time a message is sent. For example, if application processes can migrate between computers, then the mapping from logical name to physical address must be made every time a message is sent.

The use of logical names in the writing of the processes disguises the distributed nature of the control system's hardware. The difference between inter-computer and intra-computer communication is removed and thus the distributed architecture of the control system is logically united.

2.3.2 The distributed kernel level

Each computer has its own kernel, and these together form the distributed kernel level. A kernel provides the runtime environment needed to support local application processes. It provides routines for driving devices, and for creating and destroying processes, and it performs multiprogramming and virtual memory management.

All of these services effect local resources only and so they are performed autonomously by the individual kernels. Any access to the services provided by remote kernels, for example to create a process on a remote machine or to access a remote device, is coordinated by application level software.

It is common practice for lengthy kernel routines to be performed by 'executive processes' that are multiprogrammed along with the application processes. Lister (1979) suggests executive processes for performing I/O operations, and Brinch Hansen (1973) used them to perform process-creation commands. Allworth (1981) and Harper (1982) take this idea to its logical conclusion and suggest implementing every interrupt routine as an executive process.

A kernel is composed of a number of processes. One of these processes, the nucleus, is permanently locked in memory and runs in privileged mode. The other processes are the executive processes. The partition of labour between the kernel's nucleus and its executive processes is invisible to the application processes.

The nucleus provides such services as the first level interrupt handler, synchronisation primitives, and the dispatcher (Lister 1979). Communication between the nucleus and the executive processes, and communication between executive processes, will be based on a mechanism more basic than that used for communication between application processes;

for example, shared data with semaphores for synchronisation is a common choice.

Another service provided by a kernel is that of interprocess communication. In ADDAM and in the Signal Processing System, all interprocess communication is performed by an executive process. In CONIC, DSN and the Distributed Processes System, interprocess communication between processes on different computers is performed by an executive process, while local interprocess communication is performed by the kernel's nucleus. DSN can also be configured so that both local and remote interprocess communication is performed by the kernel's nucleus.

Remote interprocess communication requires cooperation between the sending and the receiving kernel. This cooperation is achieved by using the inter-computer communication mechanism provided by the hardware level.

2.3.3 The hardware level

This level consists of the computers, and of the communication network that links them together. Each computer is connected to the communication network by a network interface. Input/output devices are attached to individual computers.

Nearly all of a computer's operations are performed autonomously by that computer, for example reading and writing to local memory. However to transfer data from one computer to another requires the cooperation of both sender and receiver. This cooperation is performed using a low-level protocol provided by the hardware/firmware in the network interface.

The composition of the hardware is determined by a large number of factors, including the environmental requirements, the required performance, and to a large extent the concept of its inventors. In the above examples the trend is to use mini-computers or micro-computers linked together by some form of local area network (Clark et al. 1978, Gee 1983).

2.4 Summary

The organisation outlined in the previous section is not limited to distributed computer control systems only. A number of so-called distributed computer systems exhibit this organisation, for example DCS (Rowe et al. 1973), SODS (Sincoskie and Farber 1980) and Roscoe (Solomon and Finkel 1979).

In Section 2.1 we presented a list of five advantages that result from basing a control system on a distributed architecture. If the full potential of all of these advantages is to be exploited then the software must be designed accordingly. This is particularly true of communication and availability.

In the same way that different types of computers can communicate with each other over a common communication network, application processes written in different languages must be able to communicate with each other. In DSN application processes written in different languages with different intercommunication primitives can communicate with each other because all of the different interprocess communication mechanisms are implemented using asynchronous message passing. Similarly, if the computers are of different types then messages must be translated from one representation to another; for example, in DCLN (Liu and Reames 1977) all messages are translated into an intermediate format before being sent over the network.

As indicated in Section 2.1, possibly the greatest advantage of the distributed architecture is its high availability. This advantage is the rationale for the development of process survivability, and the rest of this thesis is dedicated to describing existing systems that exhibit process survivability and to developing process survivability for the distributed computer control system PROSUR.

3. An Introduction to Fault Tolerance

3.0 Introduction

It is impossible to build a computer system in which a fault does not exist and in which a fault cannot arise: its hardware will degrade with time, and its software will probably be too complicated to be fault free. To achieve high reliability a computer system must be able to continue operating to specification despite the presence of a hardware or a software fault. Such a system is called a fault tolerant computer system.

This chapter introduces the principles of fault tolerance, and illustrates them with two extended examples: the recovery block scheme for implementing software fault tolerance, and the hardware fault tolerant STAR (Self Testing And Repair) computer.

3.1 A vocabulary for discussing fault tolerance

A computer system is formed by superimposing a software system on a hardware system. In turn both systems are composed of sub-systems, and so on. To be able to discuss fault tolerance in computer systems we must first be able to describe the organisation of the systems themselves. The following system model reflects the structuring that should exist in all computer systems, both in their hardware and in their software. The model is based on those presented by Watson (1983) and by Anderson and Lee (1981 and 1982). It is also similar to the model proposed by Jones (1978).

A system provides its environment with controlled access to its resources. These resources may either be physical or logical; for example, disk blocks or files.

The services provided by a system are defined by a set of data structures (its representation or external state) and by a set of operations that can be performed on the system's external state. A system's behaviour is described by the effects that the operations have on its external state. The implementation of the services and the actual organisation of the resources are hidden from the environment.

Requests for operations to be performed by a system are made by the environment via the system's interfaces. An interface is simply the place of interaction between two systems. The system's environment is also a system and may in turn be composed of a number of other sub-systems.

For an example we turn to the distributed computer control system model presented in the previous chapter. An application process is a system, and its input and output channels are its interfaces. An application process's representation and operations will depend on the service it provides. If for example an application process provides access to a disk then its representation may be that of a set of files, and its operations may include commands such as open file, close file, etc. Requests for operations will be in the form of messages, and will be made via the application process's channels; the results will be returned in a similar fashion. The way in which the filestore is organised is hidden from the process's environment.

A system consists of a set of components and a design. The components provide the operations needed to implement the system's own operations, and the design defines and controls the interactions between the components, and between the components and the interfaces. If a system has no discernible internal structure, or if that structure is of no consequence, then that system is said to be atomic.

A component is itself a system. It provides a set of operations and a representation of its resources. To distinguish between the operations provided by a system and those provided by a component we call the latter actions. A system operation is implemented by a set of actions instigated by the design. The set of components' representations comprises the system's internal state. The system's representation is an abstraction of its internal state.

The design is that part of the system that receives requests for operations from the environment and in turn instigates the necessary actions to perform the operation. When necessary, the design also controls the interaction between components, and the interactions between components and the interfaces in order for the components to use services provided by the environment.

The design is a system. The system's state does not include the design's external state, as the system's state reflects the activity in the system and not its organisation, which would normally be fixed. The design should not be confused with the plan that describes a system's organisation, nor with the process by which the system was designed.

A particularly classic example of the partition of labour between design and components is given in Chapter 4, Section 4.2.1, which describes how masking redundancy can be implemented within the application level of a distributed computer control system so as to make it crash tolerant. Returning to our earlier example of a system, an application process's code is the design and its data structures are the components. The code receives requests for operations (messages) and instigates the actions to manipulate the data structures. The internal state of the application process will be the contents of the data structures. The state of the process's code (its design) is static and does not form a part of the process's state.

This system terminology can be used to describe a computer system recursively, or more conventionally, it can be used to describe the computer system in terms of a hierarchy of levels. Each hierarchical level consists of a number of systems. Each level provides the higher levels with a set of services, and in turn its systems are implemented using the services provided by the lower levels.

Since 1972 a project led by Professor B.Randell at the University of Newcastle upon Tyne has been investigating reliability in computer systems. As part of its wider research this project has been developing a vocabulary for discussing system reliability, with the aim of getting this vocabulary accepted as a standard. The final version of this vocabulary has been presented by Anderson and Lee (1981 and 1982). We have already drawn on this vocabulary for the description of a system and we now present those terms that describe reliability itself.

To be able to decide whether a system is reliable it is necessary to be able to recognise when it fails. In order to be able to do this it is necessary to assume the existence of an authoritative specification which defines a system's acceptable behaviour. If the system's behaviour contradicts its authoritative specification it is said to have failed.

The reliability of a system is often measured by the probability that no failure will have occurred by a certain time. A common alternative measurement to this is the system's mean time between failures.

A system's behaviour is defined by the effect that a given operation has on the system's external state. The authoritative specification defines what constitutes a valid external state and defines how an operation will map the external state from one valid state to another. A system fails when an operation results in a transition from a valid external state to an invalid or erroneous one.

As the external state is an abstraction of the system's internal state, an erroneous external state must mean that the internal state is erroneous as well. That part of the internal state that is actually erroneous is called the error.

An erroneous internal state is the result of an erroneous transition that changed a valid internal state into an erroneous one. An error need not cause the system to fail immediately, and a number of operations may be performed before the error manifests itself in the system's external state and causes it to fail.

The erroneous transition must be due to either the failure of one or more of the system's components or to the failure of the system's design. This is logical as a system consists only of its components and its design, and so its failure must be due to a failure in one of its constituent parts.

As a component (or a design) is a system itself, its failure must be due to an error in its internal state. To distinguish between an error in a component/design from an error in the system, the former are referred to as component/design faults. Design faults and component faults are collectively called system faults.

A fault within a component or a design will eventually cause it to fail, and this will result in an erroneous transition (the manifestation of the fault) being performed on the system's internal state, which will inject an error into the system's internal state. Further valid transitions will lead to the system failure.

3.2 Fault tolerance and fault intolerance

There are two complementary approaches to producing a highly reliable computer system: fault prevention and fault tolerance.

Fault prevention techniques attempt to produce a system in which there are no faults. Fault prevention is performed in three stages. First, techniques such as quality control, formal specification, structured programming and top-down analysis are used in an attempt to avoid introducing faults in the first place. Unfortunately in a complicated system faults are bound to be introduced, and so the second stage is to detect these faults by testing and validation, and then to remove them. Finally, having hopefully removed all of the faults, the system must be screened from external stresses in order to prevent any faults from being introduced. The software and the hardware can be protected from malicious interference by a number of security mechanisms (Shanker 1977), although it is still vulnerable to enhancements.

Fault prevention is not sufficient on its own to produce a highly reliable computer system. Hardware components deteriorate with time and they are vulnerable to external stresses that overwhelm the screening. Software is too complicated to be tested exhaustively, and program validation is still in its infancy. So despite the care taken faults will still occur and when they do the system will fail, which is why a system that relies solely on fault prevention to achieve reliability is said to be fault intolerant.

As faults will always occur, the only way to achieve high reliability is for a system to be designed to be fault tolerant. To achieve fault tolerance a system must include mechanisms that can detect a component/design fault and intercede to correct that fault before it causes the system itself to fail.

Fault tolerance does not make fault prevention redundant, but it can be used to reduce the amount of effort that is put into fault prevention, as any residual faults will be masked by the system's fault tolerance mechanisms. However, this does not mean that fault tolerance can be used as a 'warm blanket', as without any attempt at fault prevention the system would most likely not work at all.

3.3 The four phases of fault tolerance

Fault tolerance mechanisms are intended to prevent faults from causing system failures. The operation of fault tolerance mechanisms can be generalised into the following four phases:

1) Error detection

To be able to tolerate a fault the system must first be able to detect it. By definition a fault is part of a component's/design's internal state and so cannot be detected by the system; neither can the resultant component/design failure be detected as that is an event. However, the fault will manifest itself as an error in the system's internal state and this can be detected. The first stage of fault tolerance is to detect the error before it causes the system to fail.

2) Damage confinement and assessment

An error must be detected as soon as possible. Any delay may lead to erroneous information being spread around the system. If immediate detection of an error cannot be guaranteed, then it is necessary to discover the extent of the damage before dealing with the error.

Damage confinement measures may be incorporated into the system in order to limit the amount of damage that can be caused. Damage confinement measures reduce the amount of work that must be done in damage assessment, thereby reducing the runtime overheads involved in

fault tolerance.

3) Error recovery

Error recovery techniques are designed to transform the system's current erroneous internal state into an error free and well defined state from which normal service can continue.

If the types of faults that can occur can be predicted, and if the limits of the resultant damage can be accurately assessed, then specific recovery routines can be provided that will correct only the damaged part of the system's state. This technique is called forward error recovery. The techniques such as Hamming codes used in computer communication to detect and correct errors in received data are an example of forward error recovery (Cole 1982).

Forward error recovery is very efficient as the minimum of adjustments are made to the system's state. Unfortunately its reliance on being able to predict both the fault and the resultant damage limits its general application as it means that it cannot cope with unpredictable errors. However, it is very useful when conditions allow its use.

If the faults and the resultant damage cannot be predicted then the only viable method of recovery is to restore the system to an error-free past state. This technique is called backward error recovery.

The automatic repeat request techniques designed to provide error free computer communications are an example of backward error recovery. Here data is transmitted in blocks. If the receiver detects an error in the current block it discards the block, and requests the sender to repeat it (Cole 1982).

The two recovery techniques complement each other, and can be used together effectively. Two mechanisms for achieving software fault tolerance are exception handling (Goodenough 1975, MacLaren 1977, Cristian 1980) and recovery blocks (described in Section 3.6). Exception handling is based on forward error recovery and recovery blocks are based on backward error recovery. Melliar-Smith and Randell (1977) propose that exception handling and recovery blocks can be used together: recovery blocks for coping with unexpected software faults, and exceptions for coping with predictable failures in input data and operators etc. Exception handling is not used for software faults as that would require the faults to be predictable, and if they are predictable they should be removed.

4) Fault treatment

If the error is not to occur again, then it is necessary to repair the fault. To do this the fault must first be located, and this may be difficult as it is often hard to diagnose the fault from the damage that it causes. Once located the fault can be repaired, or the system can be reconfigured to avoid it. If the fault was transient then there is no need to repair it, as it has already gone.

In practice certain stages can be reduced or removed altogether by decisions made during the system's design stage; for example, the more effective damage containment is, the less work there is for the damage assessment phase to do.

It is also convenient to label the period before an error is detected as 'normal running', and the period after recovery has been performed as 'continued running'. Continued running is the same as normal running but it is often handy to be able to distinguish between the two phases.

3.4 Protective redundancy

All fault tolerance techniques are based on the incorporation of protective redundancy into the system. Protective redundancy is the extra hardware, software and processing power that is needed solely to ensure fault tolerance, but which is not needed by the system to execute its normal programs.

Protective redundancy can be categorised in two ways: as masking or standby redundancy, depending on how it is used; and as hardware, software or time redundancy, depending on what it is.

1) Masking and standby redundancy

In masking (or static) redundancy, redundant components operate in parallel to a component's operation in order to mask the effects of any failure that may occur in that component. As long as this redundancy remains effective any failure in the component will be hidden from the system's environment. Possibly the two best known examples of masking redundancy are Triple Modular Redundancy with voting (Mathur and Aviziensis 1970), and the forward error correction techniques mentioned earlier.

In standby (or dynamic) redundancy, spare functionally identical components are incorporated into the system so that when an active component becomes faulty one of the standby components can be switched in to replace it. Such a scheme presupposes the existence of the necessary error detection and error recovery mechanisms. If this reconfiguration can be done automatically by the system then the system is said to be self-repairing. A special case of this is where there are no standbys, and the system is reconfigured into a degraded system. For example, in the event of a permanent processor failure, the multiprocessor COPRA (Meraud et al. 1976, Meraud and Lloret 1978, Meraud et al. 1979) can reconfigure the software to run on the remaining

processors.

Probably the first operational self-repairing computer was the hardware fault tolerant JPL-STAR computer described in Section 3.7. The software fault tolerant recovery block scheme is also self-repairing as in the event of a fault arising in the current software module, an alternative can be used to replace it.

A third option known as hybrid redundancy is a combination of masking redundancy and standby redundancy. Masking is used to provide fault tolerance and standby components are available to replace the masking redundancy components when they fail. The prime example of hybrid redundancy is n-modular redundancy, which is triple modular redundancy with spares. N-modular redundancy is used in the Fault Tolerant MultiProcessor computer (Hopkins and Smith 1975).

Masking and standby redundancy both rely on the faults in their redundant components and in the components they are protecting being independent of each other, otherwise the fault could not be masked. For example, software fault tolerance can be provided by N-version programming which is based on masking (Chen and Aviziensis 1978, von Linde 1979), and by the recovery block scheme which is based on standby redundancy and self-repair. In both cases this protection would be nullified if the redundant software was identical to that being protected as any algorithmic fault would then be contained in the redundant and the protected software. Hence in these examples the redundant software is coded independently.

2) Hardware, software and time redundancy

Hardware redundancy is primarily to provide hardware fault tolerance, although it can also be used to provide support for software fault tolerance. Software redundancy consists of the extra data and code needed to support the four phases of fault tolerance.

Time, or program execution, redundancy is the spare processing power that must be available if the extra processing required by the addition and performance of fault tolerance is not to result in a degraded response time.

3.5 The hardcore

In the final analysis all fault tolerant systems are totally dependent for their correct functioning on some critical components that are themselves fault intolerant. These components are known as the hardcore. If any confidence is to be placed in the system's ability to tolerate faults then the hardcore that supports the fault tolerance mechanisms must be extremely reliable as 'the chain is only as strong as its weakest link'.

To achieve high reliability in the hardcore the techniques of fault prevention must be applied to it. If these techniques are to be effective the hardcore must be kept as small and as simple as possible.

In a similar vein, efforts to produce secure systems have led to the development of 'security kernels' (Popek and Kline 1978). Security kernels are small minimal kernels that, in addition to the normal tasks of a kernel, are responsible for providing the security mechanism used by the whole system: hence their correct (and secure) operation is essential for the correct operation of the whole system.

McDermid (1980) has suggested that this idea could be used to produce a 'reliability kernel'. Such a kernel would provide the recovery mechanisms used by the software mechanisms to provide software fault tolerance. By keeping the kernel small, the chances of it being implemented reliably are increased.

3.6 An example: the recovery block scheme

3.6.0 Introduction

In 1972 a project was established at the University of Newcastle upon Tyne, under the sponsorship of the Science and Engineering Research Council, in order to investigate "the utility of computer architecture and programming techniques which will enable a system to have a very high probability of continuing to give a trustworthy service in the presence of hardware faults and/or software errors" (Randell 1975).

One of the major results of this project is the recovery block scheme for providing software fault tolerance. The recovery block scheme is a general purpose mechanism for incorporating software fault tolerance into a process so that it can operate reliably despite residual faults in its code. Current work includes incorporating recovery blocks into a distributed naval command and control system based on Mascot (Anderson and Moulding 1982).

Descriptions of the recovery block scheme are given in numerous papers originating from Newcastle. Two of the best known of these are Randell 1975 and Randell et al. 1978. Also, nearly all of the other papers referred to in this section contain a description. The most all-inclusive description is given in Anderson and Lee's book (1981).

3.6.1 An overview

The reliability of a particular task within a process can be improved by implementing it as a recovery block. The syntax of a recovery block is given in Figure 3a, and an example of its use to sort an array A is given in Figure 3b.

```
Ensure  <acceptance test>
  By    <primary module>
      Elseby  <alternative module 1>
      Elseby  <alternative module 2>
                .
                .
      Elseby  <alternative module n>
      Else    Error;
```

Figure 3a

```
Ensure  A[i-1] <= A[i]   For i=2, 3 .. n
  By  QuickSort (A)
      Elseby  BubbleSort (A)
      Elseby  StraightInsertion (A)
      Else    Error;
```

Figure 3b

A recovery block consists of a primary module and an ordered sequence of standby spares - the alternative modules. Each module performs the same task (sorts the array in our example) but in a different way.

When the recovery block is entered the primary module is executed. If the primary module fails then the process's state is reset to the state that existed prior to entering the recovery block and the first alternative module is executed. (The way in which a failure is detected is described below in Section 3.6.2.) This is continued until either all of the modules have been tried and have failed, or until one of the modules has been completed successfully.

In our example QuickSort will be executed first. Then if QuickSort fails the process will be reset, thus undoing the effects of QuickSort, and BubbleSort will be tried. If BubbleSort fails then StraightInsertion will be tried.

The recovery block scheme provides software fault tolerance by incorporating redundant versions of a module into a program and then performing self-repair in the event of a module failure. This self-repair is supported by error detection and backward error recovery.

Various implementations of recovery blocks are described in Horning et al. 1974, Anderson and Kerr 1976 and Shrivastava 1978. In order to improve the efficiency of recovery blocks, hardware support for these implementations has been suggested. Two such proposals are presented in Lee et al. 1980 and Kant 1983 (the former of which has been implemented).

3.6.2 Error detection

After a module (primary or alternative) has been executed the user defined acceptance test is performed, to determine whether an error has occurred. Normally an acceptance test will test the correctness of the module's results, as more detailed testing of all of the process's variables would result in larger runtime overheads; in our example above the acceptance test checks that the array A has been sorted. If the acceptance test is passed then it is assumed that an error did not arise. The acceptance test is the embodiment of the authoritative specification.

The modules utilise the underlying operating system and hardware to perform their instructions. If such an instruction fails, for example due to division by zero, then an error has occurred, and the current module is aborted and recovery is performed.

Recovery blocks can be nested. If all of a recovery block's modules fail then the recovery block has failed, and the surrounding recovery block module is aborted.

3.6.3 Damage assessment and error recovery

Error recovery is by backward error recovery: the process's state is restored to the value it had immediately prior to entering the recovery block. In doing this all of the effects of the faulty module are removed; it is as if the module had never been performed.

Because recovery consists of rolling back the state there is no need to perform damage assessment as all alterations performed by the faulty module have been undone.

3.6.4 Fault repair

After an error has been detected and recovered from, the next alternative module is tried. This replacement is not permanent as it is assumed that the program's residual faults will only manifest themselves rarely and then due to unusual circumstances.

All of a recovery block's modules are functionally identical as they are designed to perform the same task and must pass the same acceptance test. However, they cannot be copies of each other as this would mean that they all contained the same fault. A requirement for successful standby redundancy is that the copies have independent faults, and so each module must be independently implemented.

The primary module will be the most efficiently implemented (relative to some measurement) and hence the most complicated and error prone. The alternatives will normally be ordered by decreasing efficiency, and increasing simplicity and reliability. By only temporarily replacing the primary module it is ensured that the task is performed efficiently

except for when an error is detected.

Removing the fault from the module's implementation is left to a human operator. The recovery block scheme records the frequency with which each module fails so that the operator knows which modules are faulty and which are worth repairing. If the frequency is low and the alternatives always mask the fault then it may not be worth repairing a faulty primary module.

3.7 Another example: the JPL-STAR computer

3.7.0 Introduction

The STAR (Self Testing And Repairing) computer is the result of an investigation into hardware fault tolerant computing conducted throughout the 1960's at the Jet Propulsion Laboratory (JPL) at Pasadena, California. STAR is an experimental general purpose computer that exploits standby redundancy and self-repair to achieve high reliability. It became operational in 1969.

The work on STAR was sponsored by NASA, and so its characteristics are those that would be required for an on-board computer of an unmanned spacecraft on a ten year flight. The reliability requirements for such a computer are obviously high because of the consequences of a computer failure combined with the long mission time and the impossibility of maintenance.

Work on STAR was terminated in 1972 when JPL turned its attention to designing a second-generation fault tolerant spacecraft computer based on a distributed computer system (Rennels 1978). This progression was motivated by advances in digital circuit technology which nullified the need for some of the design restrictions that had shaped STAR, and which made this more attractive solution feasible.

STAR is designed to tolerate hardware faults, but not software faults. Fault tolerance is based on hardware-implemented error detection, hardware-implemented and software-implemented self-repair, and software-implemented backward error recovery. It is of particular interest as it supports a form of process survivability.

3.7.1 Hardware and software organisation

STAR's hardware (Aviziensis et al. 1971) consists of a standard configuration of functional units supplemented by spare units that can be used to replace failed operational units. The central processing unit's functions are partitioned into a number of special purpose processors (arithmetic, logical, input/output, etc.). Memory is composed of a number of read only memory units and a number of read/write memory units. The processors and the read only memory units have dedicated spares and the read/write memory units have a pool of spares. All of the functional units are linked together by two data buses and a control bus. Figure 3c illustrates this layout.

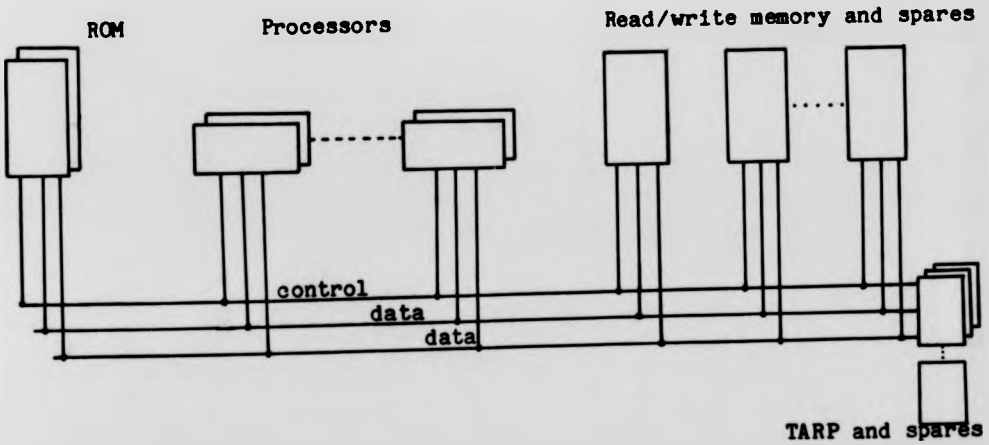


Figure 3c

The heart of STAR's fault tolerance is the Test And Repair Processor (TARP). The TARP is connected to all three buses and to each functional unit separately (the latter is not shown in the diagram). The TARP monitors the rest of the computer in order to detect any errors.

Software is divided into user processes and the STAREX executive which supports them (Rohr 1973). STAREX multiplexes the user processes and provides them with routines for performing input/output etc. It also provides the backward error recovery mechanism.

3.7.2 Error detection and fault repair

Error detection, and the majority of the fault repair, is performed by the hardware; in particular, by the TARP.

The TARP detects errors by checking the validity of every word sent over the data buses (all data and instruction words are encoded in an error detection code), and by checking status messages sent from the functional units. The TARP is thus the embodiment of the authoritative specification within the system.

When the TARP detects an error it stops the computer. The TARP assumes initially that the fault is transient and control is transferred to the STAREX recovery routines to perform backward error recovery. (In Meraud and Lloret 1978 it is reported that 90% of the faults arising in COPRA are transient.) If the fault does not respond to this treatment then the TARP classes it as a permanent fault. The TARP replaces the failed unit and then control is again passed to the recovery routines.

The TARP is STAR's hardcore. It must be reliable and so it is implemented using n-modular redundancy of TARP units.

3.7.3 Backward error recovery

As they run, both the user processes and STAREX establish recovery points. Backward error recovery restores the software component interrupted by the fault to the state that it was in when it performed its last recovery point.

When a fault has been detected by the TARP, control is transferred to the STAREX recovery routines to perform backward error recovery. They first determine which software component was interrupted and then restart it from its last recovery point. If the fault interrupted the execution of a STAREX routine then STAREX (and not the calling user process) is rolled back to its last recovery point; STAREX is relied upon to complete the routine and so a recovery point is established at the start of every routine.

User processes establish recovery points using a STAREX routine which stores a copy of the process's state within STAREX. The stored state consists of specified variables, processor register contents and the process's start address. Process checkpoints are double buffered so that in the event of a fault occurring during the establishment of a checkpoint the previous checkpoint will be intact and can be used. Processes have only one outstanding recovery point each.

STAREX itself has a high frequency of establishing recovery points and so the task is made as efficient as possible. STAREX is stored in duplexed memory (reads and writes are performed in parallel) and establishing a recovery point is simply a case of storing the restart address. After a fault the processor registers are flushed and so STAREX is programmed using the convention that all register values needed after a recovery point are stored prior to the recovery point and then reloaded afterwards.

When STAREX is rolled back to its last recovery point its data is not re-set to the values it had when the recovery point was established. STAREX will repeat code but the effects of performing that code the first time have not been undone. To prevent this resulting in unfortunate side-effects the recovery points are inserted so that the inter-recovery point code is idempotent.

3.7.4 Process survivability in STAR

If a read/write memory unit fails permanently it is replaced by one of the spares. The software that had been resident in the failed unit must be recreated in its replacement. Again this is done by the STAREX recovery routines.

User processes use simplex memory. A process is recreated from its code and constants, which are stored in the read only memory units, and from the data stored for its last recovery point. Thus all of the user processes that were using the faulty unit will have been rolled back to their last recovery point (irrespective of whose execution was interrupted).

The executive is duplexed. When one of a pair of duplicated memory units is replaced the contents of the survivor are copied into the replacement. Survivability is the prime reason for duplexing the executive as without the duplexing it would be necessary for full checkpoints of the executive to be stored on backing store.

We return to process survivability in STAR in Chapter 4, Section 4.3.2.

3.8 Fault tolerance and process survivability

When a computer in a distributed computer control system crashes the application processes that were running on that computer are lost. These application processes may be vital to the correct running of the application level and without them the control system will fail. Process survivability ensures that no application processes are lost in a crash thereby making the application level crash tolerant.

Process survivability is achieved by introducing redundant copies of the application processes into the application level. Inactive backup copies of every application process are dispersed amongst the computers. At intervals the active copy of an application process will establish a recovery point. When an application process establishes a recovery point all of its backups are updated to become exact copies of the active process. When an application process's host computer crashes one of its backups will be activated to replace it. The backup will be started in the state that its previous generation was in when it performed its last recovery point. This is similar, in general but not in detail, to the way that user processes are recreated in STAR after a permanent memory unit failure. Another similarity with STAR is that process survivability does not support software fault tolerance.

Process survivability is achieved by incorporating redundant, standby spares of every application process into the application level and by performing self-repair by activating the appropriate standby copies in the event of a crash. This self-repair is supported by error detection in order to detect the crashes, damage assessment to determine which application processes were lost in the crash, and finally by backward error recovery to restart these application processes from their last recovery point.

The software redundancy is supported by time (program execution) redundancy, and ultimately by hardware redundancy. Time redundancy is necessary if the control system is to maintain its response time. Time redundancy is needed during:

a) Normal running, to support the work that is performed to implement process survivability, for example establishing the recovery points.

b) Continued running, so that after a crash the extra work being performed by a computer due to activated backups does not affect the response time of all of its resident application processes.

Ultimately this time redundancy must be paid for by extra computers that would not be needed if the application level was not crash tolerant. Adding extra computers to support process survivability is an aspect of the 'enough' principle mentioned in Chapter 2, Section 2.1.

All but one of the following chapters in this thesis describe process survivability. First however, in the next chapter we review existing mechanisms for making the application level of a distributed computer control system crash tolerant.

4. Crash Tolerance in Distributed Computer Control Systems

4.0 Introduction

When a computer crashes it results in the instantaneous and permanent loss of that computer. Such a crash may be caused by an external stress such as a fire or by the spontaneous failure of an internal hardware component. In later chapters we will discuss partial computer failures with respect to process survivability.

The loss of a computer will not prevent the other computers from operating normally. Interconnecting the computers by a local area network allows the computers to be physically dispersed thereby providing passive protection from external stresses by limiting the number of computers that are likely to be affected.

A computer crash will also result in the loss of components from the two software levels of a distributed computer control system: the distributed kernel level and the application level. The ability of the control system to tolerate a crash will depend on how vital the lost kernel is to the distributed kernel level and how vital the lost application processes are to the rest of the application level.

In the examples presented in Chapter 2 the individual kernels operate completely autonomously except when performing interprocess communication between processes on different computers. As long as none of the kernels provide the other kernels with a unique service then the distributed kernel level will be able to continue operating normally despite the loss of one or more kernels.

The application level will contain some application processes whose loss would not prevent the control system from fulfilling its role, or whose loss would have only a marginal effect. An example of the former is

an error logger. On the other hand if the crashed computer hosted application processes that are vital for the continued operation of the surviving application processes then, despite the fact that the majority of the control system is intact, the control system will no longer be able to fulfil its role. Despite its multi-computer architecture the control system is still vulnerable to the loss of a single computer.

Process survivability makes the application level crash tolerant by ensuring that no application processes are lost in the crash. As was described in Chapter 3, Section 3.8, process survivability is achieved by incorporating redundant standby copies of every application process into the application level. In this chapter we review established methods of implementing process redundancy in a distributed computer control system.

A prerequisite for making the application level crash tolerant is that the two lower levels that support it are also crash tolerant. Fortunately, the distributed kernel level can be made so, and the computer components of the hardware level are so naturally. Unfortunately, the network, which is arguably the most important hardware component, is often the most vulnerable. If the network fails, the control system will be partitioned into a number of small isolated groups of computers, with each group acting as if it were the only survivor. We will return to the problem in a later chapter, but for the present we assume that the network is un-partitionable.

4.1 Process redundancy, an introduction

By introducing redundant copies of the application processes into the application level and by placing these copies on different computers, the application level can be made crash tolerant. There are two ways of organising process redundancy:

1) Masking redundancy

All of the copies of the application process are active. Requests are issued to any of the copies and are performed by all of them. When one copy is lost the others continue to provide the service.

2) Standby redundancy

Only one copy of the application process is active; the others are inactive backups. At intervals the backups are updated so that their states are the same as that of the active application process. When the active copy is lost one of its backups is activated to replace it.

As a third alternative Casey and Shelness (1977) suggest that their Domain Structure could be used to "exploit the inherent redundancy of the system in case of failure". However, no further details are given, and this aspect of the Domain Structure does not seem to have been developed.

In the next two sections we describe in detail how masking and standby redundancy are performed. Examples of existing systems that use these mechanisms are given.

In most of the examples we see masking and standby redundancy being applied to application processes, but there is an example of masking redundancy being applied to executive processes as well. In the introduction to this chapter we stated that if the distributed kernel level is to be crash tolerant then it is necessary that none of the kernels should provide a unique service to the other kernels. This is not always possible and an alternative is to implement the executive processes redundantly so that the unique service is not lost in a crash.

4.2 Masking redundancy

4.2.0 Introduction

Masking redundancy can be applied to systems of processes as well as to individual processes. In practice it appears that masking redundancy has only been used to implement server systems that interact with their users by requests and replies.

4.2.1 Masking redundancy in theory

A crash tolerant server consists of a set of replicate servers each of which is running on a separate computer. Each replicate server provides the same services, which in turn are the same as the services provided by the crash tolerant server as a whole. Requests for operations to be performed can be made to any of the replicates. In the event of a computer crash the surviving replicates continue to provide the services.

Some crash tolerant servers provide a time invariant service in that past operations do not affect future ones, for example, reading from a file. In this case each replicate can receive and perform requests for operations independently of the others. In the event of a replicate being lost in a crash its users can use one of the surviving replicates instead; it does not matter that the internal states of the replicates are different.

However, most services will be time dependent and past operations will affect future ones, for example if writing as well as reading from files is supported. If after a crash the lost replicate's users are to be able to use one of the surviving replicates instead, it is necessary that the external states of all of the replicates are identical to each other. The rest of this sub-section, and the examples in the next, are concerned with how a crash tolerant server that provides a time dependent service can be implemented.

Using Le Lann's terminology (Le Lann 1979) a crash tolerant server can be described as a system consisting of a set of producers and a set of consumers (see Figure 4a). Each producer and consumer pair would constitute a non-redundant server. Producers and consumers are each systems in their own right and are each composed of one or more processes. In many examples the roles of producer and consumer are combined.

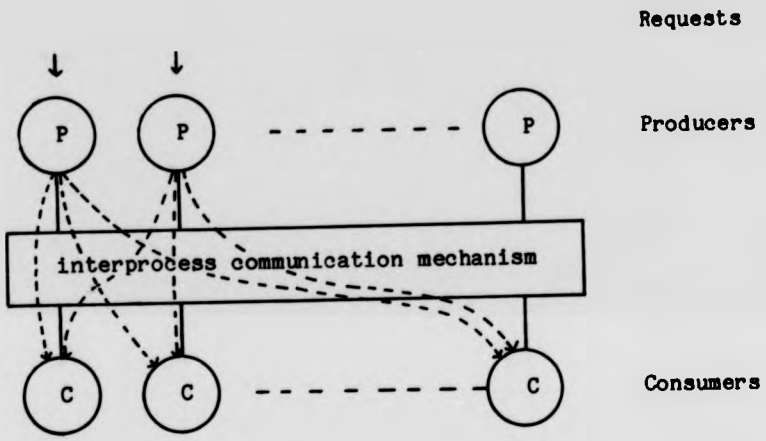


Figure 4a

A crash tolerant server (henceforth simply called a server) is a system as defined in Chapter 3. The server's service is defined by a set of operations and by its external state on which the operations are defined to operate. The server's internal state is the set of the external states of its consumers. The producers are part of the server's design and so their states do not form part of the server's state. Every consumer implements the same set of actions, and these actions are identical to the operations provided by the server: there is a 1-to-1 mapping from operation to action.

Requests for operations can be issued to any of the producers. For the server to be able to continue after a crash the external state of each of its consumers must be identical (their internal states need not be identical as they may be implemented differently). To achieve this, when a producer receives a request for an operation it 'broadcasts' a request for the corresponding action to all of the consumers.

An operation will only produce a valid result if it is performed on a consistent internal state. Consistency in a crash tolerant server is composed of:

a) Internal consistency

Each consumer's external state satisfies a meaningful predicate known as the system invariant.

b) Mutual consistency

Each consumer's external state is identical.

An operation maps the server's internal state from one consistent state to another. To prevent invalid results being produced by an operation being performed on an inconsistent state, the operations must be performed atomically. Following Le Lann 1983 we define an operation to be atomic if:

a) The actions that implement an operation are either all performed successfully or none of them are. Any results produced by the actions do not survive the failure of an operation.

b) Operations must not interfere with each other. The partial or potential output from one operation must not be used as input to another operation at the same time.

These two aspects of atomicity are known as failure atomicity and

serialisability respectively (Spector and Schwarz 1983).

Requests for operations may be received by different producers concurrently. If the producers were to implement requests in parallel then the consumers' states would no longer be mutually consistent. If two or more producers were to broadcast requests for actions at the same time, then due to variations in interprocess communication delays, the consumers would receive and obey these requests in a different order from each other. This would result in a breakdown in mutual consistency. For example, the same lock may be allocated to a different user by different consumers.

To prevent this, the requests for operations must be serialised. The serialisation of concurrent requests to a server is called synchronisation (Kohler 1981). In a crash tolerant server the aim of synchronisation is to ensure that the order in which actions are performed by each consumer is the same (Le Lann 1983).

Synchronisation can be achieved in a number of ways:

a) Event ordering

The operations (events) are time stamped by the producers, and the consumers perform the actions in that order. The time base can be provided by physical clocks, logical clocks and sequences (Le Lann 1983).

b) Voting

The producers communicate with each other to decide upon which request is to be performed next. Voting schemes include synchronous voting and majority consensus (Holler 1983).

c) Executive privilege

One producer at a time has exclusive access to all of the controllers. This privilege may be assigned permanently, in which case all requests are filtered through the privileged producer by the other

producers. Alternatively the privilege can be assigned to the producers in turn, for example by using circulating tokens or shared variables (Holler 1983, Le Lann 1983).

When synchronisation is achieved by cooperation between a group of peer processes it is called decentralised synchronisation, and control of the server is said to be decentralised. If synchronisation is based on a unique component, for example a single physical clock, then synchronisation is centralised, and control of the server is also centralised. A centralised synchronisation component with backups is not classed as decentralised (Jensen 1983, Le Lann 1979).

Masking redundancy can be based on either type of synchronisation as long as the synchronisation mechanism is implemented so that crashes do not cause it to fail. If for example a circulating token is used, then it must be possible to regenerate the token if it is lost. Furthermore this recovery must be achieved in a decentralised way that does not depend on a unique arbitrator. The synchronisation mechanism must be as resilient as the servers that use it.

It is not sufficient simply to broadcast the actions. If internal and mutual consistency is to be maintained the operations must also be failure atomic. At the start of the action each consumer establishes a recovery point. At the end of the operation if every consumer has completed its action successfully then all of the consumers discard their recovery points, but if one or more of the consumers failed to complete their action then all of the consumers roll back to their recovery points. Synchronising the rolling back of all of the consumers or synchronising the discarding of the recovery points by all of the consumers would be achieved by using a mechanism such as the two-phase commit protocol (Gray 1978).

If a consumer fails to complete an action because its host computer crashes then the other consumers will still be able to complete the operation. Were a missing consumer to result in operations being aborted then each consumer would constitute a single point of failure, which is the opposite of what is required.

When one or more components of a server are lost in a crash it is possible that the request will be lost and not be performed, or alternatively it is possible that the request will have been performed but the reply was lost. The user will have sent the request and be waiting for a reply. These two situations are inconsistent. The normal approach to restoring consistency is for the user to detect the failure of its request by a time-out on the reply, and for it to repeat the request. If the producer to which the failed request was addressed has been lost then the user must locate another producer.

If the previous request failed because the reply was lost, then repeating the request will lead to the operation being repeated. To prevent this either the operations are implemented idempotently so that they can be repeated, or the producers must be able to detect repeated requests and act accordingly.

4.2.2 Masking redundancy in practice

As mentioned in the introduction to this thesis, the current generation of command and control system used in Royal Navy warships is based on a single computer. This makes the warship dangerously vulnerable to computer loss. In 1975, in an attempt to remove this weakness, the Admiralty Surface Weapons Establishment (ASWE) instigated the DIAS (Distributed Information Architecture for Ships) programme. The aim of this ongoing programme is to develop a crash tolerant distributed computer system for use as a warship command and control system. The results of the first two phases of the DIAS programme - ADNET - were described briefly in

Chapter 2, Section 2.2.4. We now present a more detailed description of ADNET and a description of the work that has been done for the third phase.

ADNET is configured as a number of Ferranti Argus 700 military computers linked together by an ASWE Serial Highway. The peripheral devices such as weapons, sensors and interactive consoles are connected directly to the computers.

Each computer's software is comprised of a Mascot Operating System, a communications package and a number of application processes. The communication package provides interprocess communication in the form of message passing. Two types of message passing are supported: broadcast and point-to-point.

The application processes on a computer can be divided into those that support local device handling, and those that contribute to the global resources of the system. The latter consist of those processes that form part of the overall system database, and those that provide some service used by the system as a whole.

The failure of a computer must not prevent the rest of the system from functioning. It is acceptable that those application processes that are concerned with handling devices will be lost. However, those application processes that support global services must be preserved, and this is done by masking redundancy.

In ADNET application processes are classed as users, servers, producers and consumers with the usual meanings. To use a service the user first locates the server by broadcasting a 'request for service' message to each computer. The server will reply with its address, and then the user and server enter into a conversation conducted in terms of requests and replies.

Masking redundancy is only applied to application processes that are servers. So far it is only being used to implement a partitioned and replicated database called ADDAM (ASWE Distributed DAtabase Management system, Tillman 1982). Previously it had been used to implement another server (a track designator) but that server's function was later merged with that of ADDAM. The following description is based on masking redundancy as used in ADDAM.

Synchronisation is achieved by organising the server replicates into a master-slave relationship. Requests are sent to the master who performs the operation and then broadcasts the request to the slaves. The master then replies to the user. The master prevents concurrent access by mutual exclusion. (The roles of producer and consumer are combined in a server.)

Failure atomicity is omitted in order to reduce overheads. Instead each request is numbered by the master and when a slave detects that it has missed a request it asks the master for an update to its state.

The master and slave servers monitor each others' health. If the master is lost then the slaves vote amongst themselves to elect a new master. There also appears to be a facility for creating new slaves in order to maintain the level of redundancy.

The loss of a master may result in the loss of a request or a reply. The user detects a failure when a reply is not returned after a certain period. The user locates the new master by again broadcasting a request for service; on being sent the address of the new master it repeats the request. The literature does not explain how the servers avoid repeating requests.

Alsberg and Day (1976) propose a master-slave organisation for providing resilient access to resources. The master and slave replicates are organised into an ordered chain with the master at its head. A request can be issued to the master or a slave, but if it is issued to a slave it will be forwarded to the master for processing. Synchronisation is achieved by passing the requests down the chain with each replicate performing them in the same order. There is no mention of failure atomicity.

The master and slaves monitor each others' health by regularly passing messages up and down the chain. If the master is lost, the first slave in the chain takes over. If a slave is lost then the chain is reconfigured to avoid it. Recovery from lost requests or replies is performed by the user repeating the request.

In Saphir (Gaude et al. 1980) masking redundancy is implemented at the computer level. The computers are paired. Each pair consists of a master and a slave computer, both of which run the same software. All of the network traffic that is directed to a pair of computers is received by the master computer who then passes it on to its slave so that it is processed by both of them. All output produced by a slave computer, whether to the network or to the devices, is discarded by the hardware. When a master computer fails, its slave computer is able to use both the network and its devices normally, and it continues to provide the pair's services.

The Honeywell Experimental Distributed Processor (HXDP, Jensen 1978, Jensen et al. 1977) is another experimental warship command and control system. From the details given in Boebert et al. 1978 it appears that HXDP is also implemented using masking redundancy, although in this example it is the executive processes and not the application processes that are implemented in this way. The distributed kernel level is crash

tolerant, and application processes are simply lost.

4.3 Standby redundancy

4.3.0 Introduction

Standby redundancy is applied to individual processes. It does not matter whether they are users, servers, producers or consumers.

4.3.1 Standby redundancy in theory

A process is redundantly implemented as an active running replicate and a number of inactive backup replicates. All requests for operations are sent to the active replicate which performs them. Each replicate is located on a different computer. When the active replicate is lost in a crash one of its backups is activated to replace it. The decision as to which backup to activate can be based on some fixed ordering between the backups, or on voting between the backups. However this decision is made its implementation must be crash tolerant, for example if an ordering is used then it must be resilient to backup loss. After a crash, those processes that were communicating with the lost replicate must be able to locate its reincarnation.

Again it is possible to distinguish between processes that provide time dependent and time invariant processes.

When the service provided is time invariant it would be acceptable for the backups' internal states to be left in their initial states. This would be very economical, although in this situation it would be even more economical to use masking redundancy instead as then the work load could be shared between the replicates.

Where the service is time dependent however, the backups' internal states must be the same as the internal state of the active replicate so that the activated backup can carry on where the previous incarnation of the process left off. As in the previous section on masking redundancy we will concentrate on how processes that provide a time dependent service can be implemented using standby redundancy.

To update the backup replicates' internal states a copy of the active replicate's internal state must be used. It is impossible to continually update the backups' internal states and so they are updated at intervals when the active replicate establishes a recovery point.

When a backup is activated it will be in exactly the same state as its previous incarnation was when it performed its last recovery point. The backup replicate will start to execute its code from immediately after that recovery point. Because recovery points are only performed at intervals, the internal state of an activated backup will be out of date.

Restarting processes from a past state can lead to inconsistencies arising between the internal states of communicating processes. The precise details will depend on the interprocess communication mechanism used. (A full description of the effect that it has on asynchronous message passing is given in Chapter 7.) In general however, having rolled back one or both processes of a communicating pair the following inconsistencies may arise:

a) Data will have been received that has not yet been sent

An example of this is shown in Figure 4b which shows two processes at the time of a crash. (In this and later figures process execution is represented by a directed vertical line, recovery points are represented by a horizontal '[' character, and the horizontal directed lines each represent the passing of a message from one process to another.) Prior to the crash, and after its last recovery point,

Process-A sent Process-B a message. After the crash Process-A is restarted from its last recovery point, and Process-B continues normally. The recovery point was established before the message was sent and so Process-B has received a message which, with respect to Process-A's current state, has not yet been sent.

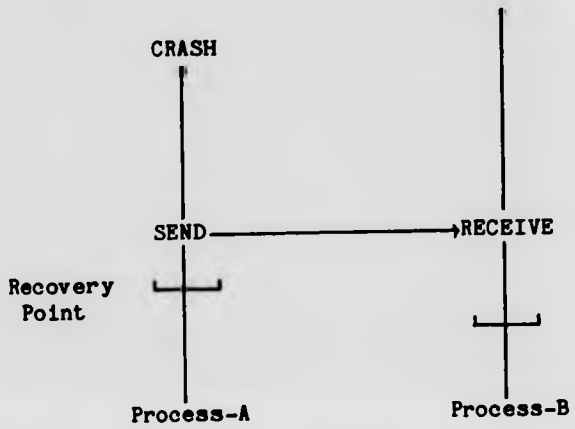


Figure 4b

b) Data that has been sent has been lost

Figure 4c below shows another two processes at the time of a crash. Prior to the crash, and after its last recovery point, Process-C received a message sent to it by Process-D. After the crash Process-D continues normally but Process-C is restarted from its last recovery point. The recovery point was established before the message was received and so after being restarted Process-C's internal state does not contain the message. The message sent by Process-D has been lost.

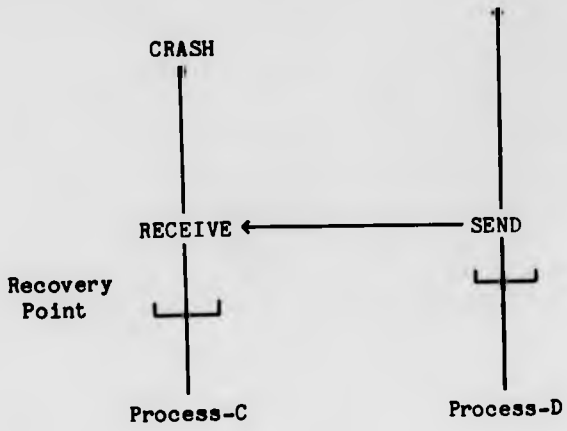


Figure 4c

These inconsistencies must be removed. The consequences of not removing them could be disastrous. For example, if an alarm message is lost the control system may not react quickly enough to a critical situation in its environment. Similarly, if a message exists that was not sent then no matter how the sender acts when it is restarted, the receiver will perform an action that it was not asked to do. Two ways of restoring consistency are available. One attempts to roll back the processes to a set of checkpoints where the processes' states are consistent with each other, and the other turns the inconsistent state into a consistent one without further rollbacks. Process survivability employs the latter.

Standby redundancy is applied to individual processes rather than to systems of processes because performing a recovery point that spans more than one process is technically difficult especially if those processes are on different computers.

4.3.2 Standby redundancy in practice

Possibly the most famous commercially available crash tolerant computer is the Tandem 16 NonStop System. The Tandem 16 was developed to provide commercial firms such as publishers and financiers with a hardware fault tolerant computer for on-line transaction processing. The prime aim of the Tandem 16 is to provide a guaranteed continuous operation despite a single hardware fault.

The following description of the Tandem 16 is based on those given in Bartlett 1978, Katzman 1978, Mackie 1978, Bartlett 1981 and Paker 1983.

A Tandem 16 can consist of up to 16 computers loosely linked by a duplicated external bus. Devices are attached to I/O Controllers. Each device can be connected to two I/O Controllers which in turn are each connected to two computers. Figure 4d shows a Tandem 16 consisting of two computers and a single multi-homed device.

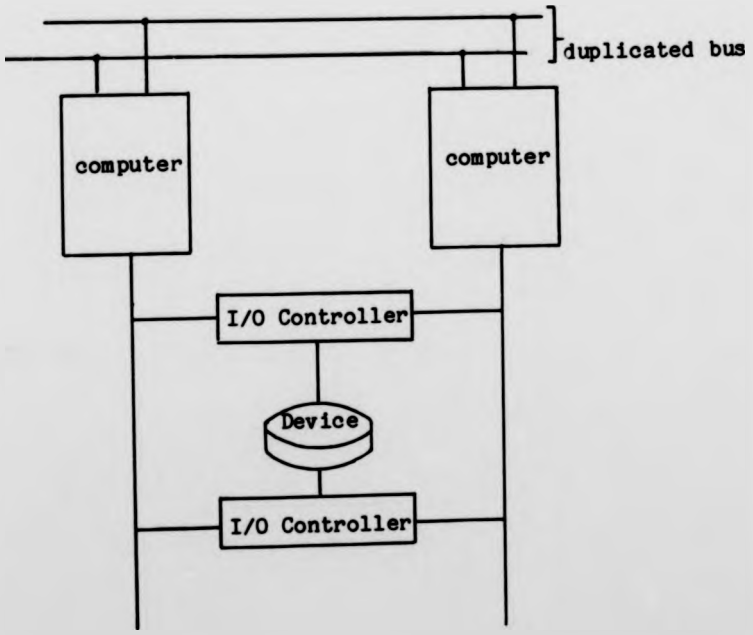


Figure 4d

This hardware arrangement removes all single points of failure from the hardware. Unfortunately the use of a bus for inter-computer communications, while providing a very high data rate of 13M bytes/second, limits the dispersal of the computers (in fact they are mounted all together in a single cabinet) thus making the entire system vulnerable to external stresses, for example a fire would easily disable the whole system. In fairness it should be noted that it was not designed to cope with such calamities as they would result in more than a single hardware fault.

The software is divided into processes. Interprocess communication is by message passing, where a message consists of a request and a reply. To provide a non-stop service the Tandem 16's software, like the hardware, must not contain any single points of failure. This is achieved by standby redundancy of processes.

Any process can be implemented redundantly as a process-pair consisting of a primary process and a single backup process. (The large number of computers in a Tandem 16 is to ensure a high through-put rather than to support low vulnerability.) At intervals the primary process establishes a recovery point which updates the backup process's internal state. When the primary process is lost in a crash the backup process will be activated and it will start processing from the point at which the last recovery point was performed.

To ensure that consistency is restored after a crash, the following protocol is used by the processes to control process interactions. This protocol is shown diagrammatically in Figure 4e.

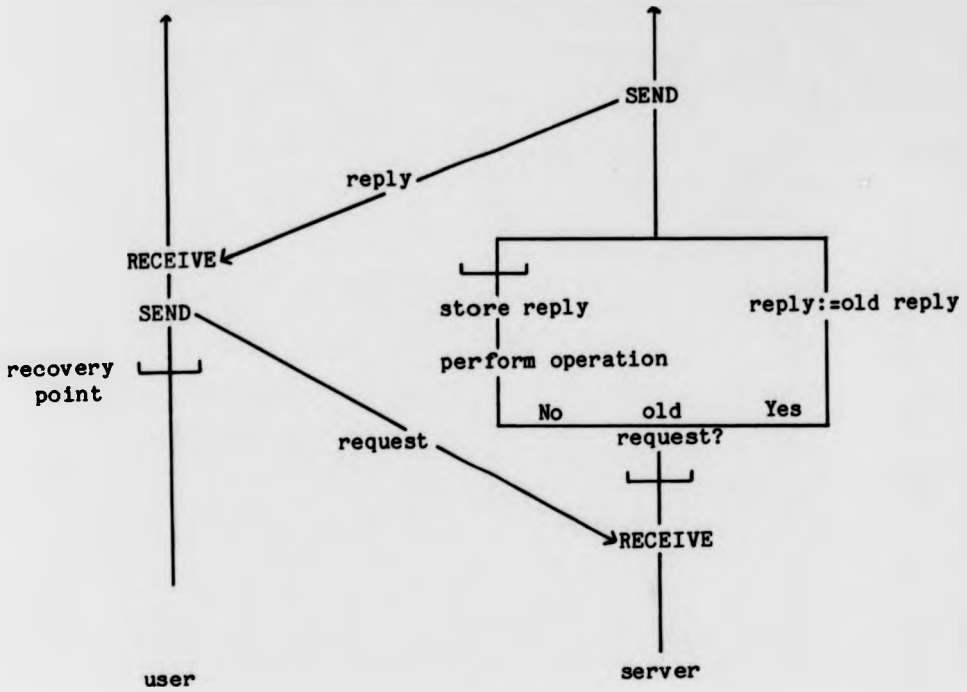


Figure 4e

Before issuing a request the user process performs a recovery point. It then sends its request to the server primary process and waits for a reply. The server receives the request, performs a recovery point and then performs the requested operation. All requests are sequence numbered, and once the operation is performed the server stores the request's sequence number and the reply. A recovery point is performed and then the reply is returned to the user.

In the event of either or both processes being lost before the transaction is completed the request will be repeated thereby ensuring that the operation is performed at least once.

If the server crashes after a request has been sent to its primary process then the message system automatically resends the request. This time the message is sent to the server's now active backup process. Depending on whether the server had established its first recovery point or not, this request may or may not be a repeat. (Even if the server had received the original request and performed the recovery point its backup would not have been able to return the reply.)

Some operations are idempotent but the majority are not. The server uses the request's sequence number to detect repeated requests. If a request is a repeat, then instead of performing the operation the server returns the stored reply. Thus the request is not repeated but the results are still obtained.

If the user process crashes after sending its request, then its backup will restart from its last recovery point and will repeat the request. The server may have performed the original request but not been able to return the reply, or alternatively it may not have received the request. In the former case the server will return the stored results as described above, and in the latter case it will handle the request normally.

Thus each process has only one outstanding recovery point, and consistency is recovered by the above protocol implemented by the user and server processes' programmers. When a recovery point is created the process (and hence the programmer) must specify which part of the data segment is to be backed up. Obviously, if recovery from crashes is to restore consistency then this protocol must be implemented correctly.

A similar problem arises in the JPL-STAR computer which uses standby redundancy to ensure that its user processes can survive the permanent failure of their host memory units. A description of process survivability in the JPL-STAR computer was given in Chapter 3, Section

3.7.4.

The user processes in STAR communicate with each other via shared data stored within the STAREX executive. Access to the shared data is provided by routines supplied by STAREX. As STAREX is housed in duplex memory the permanent failure of one of its memory units will not affect its data. Alterations made to the shared data are permanent and are not affected by memory unit failures.

If after establishing its last recovery point, and prior to a crash, a user process performed a STAREX routine, then on restarting it will repeat that routine. Some STAREX routines are idempotent and can be repeated, but others are not and these must not be repeated. To overcome the latter problem STAREX provides a number of mechanisms that enable user processes to either undo the effects of an executive routine before repeating it or to skip the repeated call altogether.

DEMOS M/P (Powell and Presotto 1983) takes the same approach to crash tolerance as process survivability does (although the implementational details are very different). Both are based on the fact that application processes are deterministic and so, by giving a restarted process the same input as it had prior to the crash it will recover itself to a consistent state. This approach has the great advantage that crash tolerance is transparent to the application programmer. This lack of transparency is a positive drawback to the Tandem approach to crash tolerance.

DEMOS M/P consists of a broadcast network, a number of computers and a computer called the 'recorder'. The recorder plays a similar role to that played by STAR's TARP in that it acts as a repository for recovery data and it controls recovery after a crash. When a process performs a recovery point a copy of its internal state is recorded within the recorder. Every message sent over the network is copied and that copy is

stored within the recorder (all interprocess communication is inter-computer).

When a process is lost in a crash the recorder causes it to be moved to a different computer and then restarted from its last recovery point. The recorder then sends the restarted process, copies of all of the messages that had been sent to the restarted process's previous generation after the last recovery point had been established. Any message produced by the restarted process and which was also sent prior to the crash by the restarted process's previous generation is discarded in order to prevent side-effects.

The recorder is DEMOS M/P's hardcore. Although it is claimed that by centralising the recovery function its implementation is simpler and hence more reliable, it does make the entire control system vulnerable to the loss of the recorder.

In the Auragen computer (Borg et al. 1983) this vulnerability is removed. As in Tandem, processes in the Auragen are implemented as process-pairs. Every time a primary process executes a recovery point its inactive backup process is updated so that both primary and backup process have the same internal states. When a primary process sends a message, copies of that message are sent to the receiver's primary process, the receiver's backup process and the sender's backup process.

When a backup process is activated it has copies of the messages that were sent to its primary process prior to the crash, and it has copies of the messages that were sent by its primary process prior to the crash. The former enable it to recover itself to a consistent state, and the later allows the operating system to detect when the backup process repeats a message that was sent prior to the crash; such messages are discarded.

The Auragen approach is a combination of the best facets of Tandem and DEMOS M/P: it is invulnerable to a single hardware failure and crash tolerance is transparent to the application programmer.

Project Little (Brenner et al.1980) was an experimental ultra reliable distributed computer system. Although not a distributed computer control system its hardware and software organisation conforms to the three-layer model presented in Chapter 2. Each computer runs a control program which is either an EXEC which supports one user process, or an SVOL which provides access to one storage device.

Little also employs a form of standby redundancy. As a user process executes, its host EXEC records information about the process's state (although not actual checkpoints) in a file called the Tasklist which is stored on more than two SVOLs. In the event of a computer crash an idle EXEC will continue the interrupted process. No further details are given in the literature, and the project was terminated before this aspect of Little could be developed (Burton 1982).

Crash tolerance by standby redundancy is very similar to software fault tolerance by standby redundancy as both require the storing of past states. Inconsistencies similar to the two described in Section 4.3.1 also arise when backward error recovery is performed by one or more processes within a group of communicating processes. It is possible that the solutions used there could be applied to crash recovery as well. Most of the work that has been done on backward error recovery amongst communicating processes has been as a result of interest in the recovery block scheme described in Chapter 3, Section 3.6.

A process's active and backup replicates would each maintain a sequence of recovery points. After a crash backups would be activated to replace the processes that were lost. Then the active processes would be rolled back to older recovery points in order to place the system into a

consistent state. The set of recovery points that satisfy the criteria constitutes the recovery line. For example, referring back to the two examples shown in Figures 4b and 4c, consistency would be restored by rolling back Process-B to its previous recovery point, and by rolling back Process-D to its last recovery point. Thus crash recovery would result in the need to roll back processes being propagated from one process to another until a consistent state is achieved. Rolling back an active process would necessitate that its backup replicates also discard the same recovery points.

Unfortunately there are two major drawbacks to this solution:

1) Data and processing overheads

Each process would consist of a number of replicates, each of which would have to maintain and store a possibly large sequence of recovery points. This sequence would increase as the process runs. However, it is possible to calculate which recovery points will never be needed and thereby delete the data that supports them. Of course this would in turn increase the processing load.

2) The domino effect

The propagation of rolling back can result in the so called 'domino effect' where a crash can result in a group of processes having to roll back over many if not all of their recovery points. Figure 4f below illustrates this effect.

If in our example the computer hosting Process-A crashes, then one of Process-A's backup replicates is activated to replace it. In effect Process-A has been rolled back to its most recent recovery point. To achieve consistency Process-B is rolled back to its third recovery point, which in turn means that Process-C must be rolled back to its second recovery point and so on. The eventual outcome will be that all of the processes will have been rolled back to their initial state.

Also, all of their backup replicates will have discarded all of their recovery points as well.

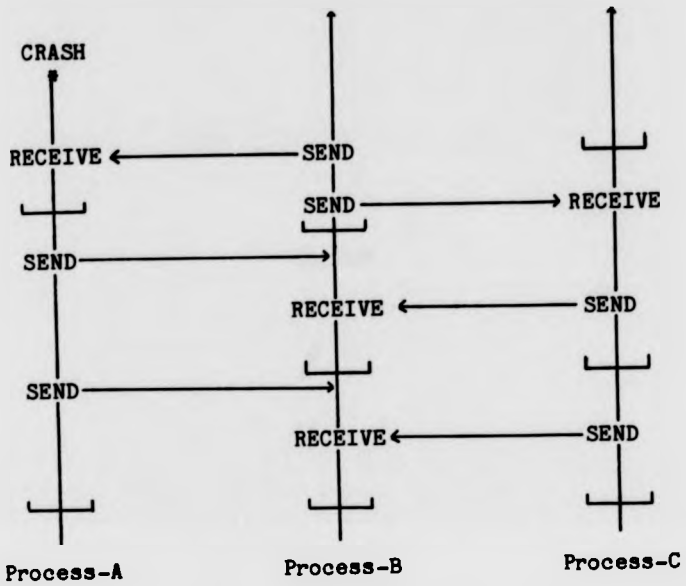


Figure 4f

The domino effect is obviously very unsatisfactory especially in a real-time system. The use of standby redundancy for ADNET was rejected because of the "immense backward error recovery problems" (Moulding 1980a), although as we have seen in Tandem and as we shall see for process survivability this is not necessarily the case.

Examples of using the domino effect for backward error recovery in multi-computer systems can be found in Merlin and Randell 1978, Menasce 1978, and McDermid 1981a and 1981b. McDermid suggests that crash recovery could be performed in this way. It has also been suggested that it could be used to provide crash tolerance in the LOCUS distributed operating system (Popek et al. 1981) and in the related UCLA-Net (Rudisin 1980).

Russell (1977 and 1980) describes a technique known as 'Directed communications' that guarantees that a recovery line can be found without having to roll back the processes too far, thereby limiting the worst of the domino effect. (It also limits the number of outstanding recovery points that are needed, thereby reducing the data storage overheads.) Russell's solution is complicated when used for error recovery because of the need to verify that the messages do not contain errors, but for crash recovery this would be unnecessary.

Propagating recovery between processes and 'Directed communications' both try to find a recovery line (which may not exist) when it is needed. The alternative approach is to ensure that one always exists, and that the minimum amount of rolling back is performed. Horning et al. (1974) describe the 'conversation' for use in multiprocess error recovery, and it is possible that it could be used for crash recovery as well, although the requirement that all of the processes involved in a conversation must leave it at the same time may, as Russell and Tiedeman (1979) suggest, "cause an unnecessary loss of efficiency in parallel processing".

Unfortunately all of these schemes involve processes that were not on the crashed computer having to roll back. Ideally only those processes that were on the crashed computer should have to roll back. On the positive side, they may be able to support crash tolerance and software fault tolerance at the same time.

4.4 Summary

In this chapter we have described two ways of organising process redundancy so that the application level (or the distributed kernel level) can be made crash tolerant. In the following chapter we introduce process survivability itself, and relate it to the two major examples - ADNET and Tandem - that were given in this chapter.

5. An Introduction to Process Survivability

5.0 Introduction

The multi-computer architecture of a distributed computer control system has two characteristics that make it a suitable base for a crash tolerant computer system. Firstly, and most importantly, the loss of one computer does not prevent the other computers from working normally. The loss of a computer will reduce the control system's overall processing power, but the majority of the power will still be available. Secondly, by basing the control system around a network the computers can be physically dispersed, thus providing passive protection against external stresses by limiting the damage that can be caused. For example, by placing the computers in different buildings, the number of computers that can be damaged in a fire is reduced.

Unfortunately, these characteristics are not sufficient on their own to make the distributed computer control system invulnerable to computer crashes. The loss of an application process may result in the control system's application software failing to meet its specifications. Despite the architectural advantages of a distributed computer control system it is still vulnerable to a single computer crash.

Process survivability removes this vulnerability. Process survivability is the implementation of an application process using standby redundancy so that in the event of its host computer crashing an application process will survive and will continue to run on a different computer.

Process survivability supports 'n out of m' crash tolerance. In a distributed computer control system of m computers, every application process has n backups. The application level can tolerate n computer crashes; after that any further crashes will result in application

processes being lost and in process survivability for the other application processes breaking down. The size of n will depend on a number of factors: the size of m, the magnitude of the envisaged threat to the control system, the cost of having what is in effect n redundant computers, and the level of acceptable performance because the overheads of process survivability increase along with the size of n.

Unfortunately, while process survivability is in our opinion a necessary requirement for a crash tolerant computer system, it is not a sufficient one. When a computer crashes those devices that are attached to it are no longer accessible. Without these devices the control system will not be able to exert full control over its environment. Providing a crash tolerant input/output service is outside of process survivability's brief, and so we have adopted the approach taken in ADNET where it is accepted that a crash will result in the loss of devices as well. We shall return to this topic in later chapters.

5.1 The aims of process survivability

A number of requirements concerning the way that process survivability relates to the application programmer and how it should be implemented were defined. These requirements then guided the development of process survivability.

The primary requirement is that process survivability should be completely transparent to the application programmer: application processes are written as they would be if non-redundant. Redundancy is only apparent when the control system is configured as then the number and location of the application processes' backups are specified. Were process survivability to be implemented in part by application process code (as is the case in Tandem and ADNET) then it would be impossible to guarantee successful crash tolerance, as any mistake on the part of the programmer

could result in process survivability failing. By making process survivability transparent we can guarantee that once the process survivability code is debugged it will always work correctly.

Process survivability is based on standby redundancy rather than on masking redundancy because we believed that it would constitute the best basis for achieving transparency. Although we cannot categorically state that standby redundancy is the best basis, we can state that it has been possible to make process survivability transparent to the application programmer, and we do not believe that this would have been possible had masking redundancy been used.

Transparency would also be advantageous were it possible for process survivability to be added to an existing distributed computer control system, as transparency would mean that it would not be necessary to retrain programmers nor would it be necessary to rewrite application process code. Both of these are financially attractive, and could tip the balance in favour of adopting process survivability for an existing control system.

The drawback to process survivability being transparent is that it is impossible to take advantage of any optimisations that might arise in a particular application. However we believe that the value of such optimisations is far outweighed by the consequences of faulty crash tolerance.

The secondary requirements were guidelines as to how process survivability was to be implemented:

a) If the control system is to be effective it must be able to maintain its response time to external events. This is particularly important when an external stress has destroyed a computer. For example, a chemical works may become critical after an explosion and prompt action by the

control system would be needed to prevent a catastrophe. In order to maintain the response time we require that:

- The processing overheads of supporting process survivability during normal running should be kept as low as possible.

- The interruption to an application process's service caused by a crash should be as short as possible.

- The disruption caused to application processes that have not crashed should also be as small as possible. In particular, there should be no domino effect - application processes that are not on a crashed computer are not rolled back.

b) Each application process should have only one outstanding checkpoint. More are not needed because of our determination not to use dominoing to restore consistency.

c) Crashed application processes are recovered independently of each other. There is no coordination between the recovery of two application processes even if they normally communicate with each other.

d) Process survivability must be able to cope with multiple computer crashes even if they occur simultaneously.

e) Assuming that an application process's redundancy has not been exhausted, a process must be able to recover from a crash no matter what it is doing at the time. In particular, application processes that are currently recovering from one crash must be able to recover from another crash.

The implementation has adhered to the last four requirements, and has attempted to minimise those overheads specified in the first one.

5.2 An introduction to the implementation of process survivability

Rather than trying to make process survivability completely general and suitable for all distributed computer control systems, we have developed it for a specially devised 'paper' distributed computer control system called PROSUR (PROcess SURvivability). PROSUR conforms to the layered model of a distributed computer control system outlined in Chapter 2. Full details of PROSUR are given in the next chapter, Chapter 6.

All application processes are implemented as a process-set, where a process-set consists of an active primary replicate and a linearly ordered set of backup replicates. By having a number of backups per process-set we ensure that an application process can survive multiple crashes even when they occur simultaneously (for as long as its redundancy is not exhausted).

At intervals the primary replicate performs a recovery point which causes each of its backup replicates to be updated so that they are exact copies. When a primary replicate is lost in a crash the first of its backup replicates is activated to replace it. The activated backup replicate will start to execute from the place at which the last recovery point was performed - there is only one outstanding recovery point per process.

Chapter 6 contains a high-level description of how process-sets are organised. Full implementational details of the process-set, including how computer crashes are detected, are given in the first half of Chapter 9.

In Chapter 4, Section 4.3.1, we briefly described the type of inconsistencies that can arise between two application processes after one or both of them have been restarted after a crash. In Chapter 7, we

describe these inconsistencies in the form in which they arise in PROSUR.

In process survivability consistency is restored by replacing lost messages, and by forcing the restarted processes to act in exactly the same way as they did prior to the crash, with any repeated messages being discarded. The crux of this solution is that the process must act in the same way, and this is guaranteed by executing recovery points in the correct place. In this way consistency is restored without the need to roll back processes that were not on a crashed computer.

Chapter 8 outlines how consistency is achieved, and Chapter 9 gives full details of how it could be implemented in PROSUR.

5.3 Comparisons with existing crash tolerant systems

Of the crash tolerant systems described in detail in the previous chapter, process survivability is most similar to crash tolerance in Tandem. Both are based on standby redundancy with a single outstanding checkpoint per process and both achieve consistency without the need to 'domino' processes.

Minor implementational differences occur because of differing design aims and because Tandem is a finished product whereas process survivability and PROSUR are experimental. For example, Tandem is designed to withstand only a single component failure and so each process has only a single backup, but process survivability is designed to withstand multiple computer failures and so every process has a number of backups.

The major difference between the two is the way that consistency is restored after a backup is activated. In Tandem consistency is restored by a protocol implemented by application process code. If consistency is to be achieved then the protocol must be correctly implemented by the application programmer. In process survivability consistency is restored

by the underlying process survivability mechanism and not by application process code. Once debugged, process survivability can be guaranteed to work always, whereas crash recovery in Tandem is vulnerable to programmer error. As Bartlett (1981) says, "Many [programmers] do [write application process-pairs], and if the design has been done carefully, they will recover correctly".

ADNET's crash tolerance is based on masking redundancy and so comparing its implementation with that of process survivability is of little value. However a comparison of the services provided by each of them is worthwhile.

Both provide crash tolerance to multiple computer failures. However, in ADNET crash tolerance is only implemented for server processes (all other classes of process are lost in a crash), but process survivability is applied to all of the application processes. Thus with process survivability all of the control system's application processes survive a crash, but in ADNET only the server processes do. Finally, crash tolerance in ADNET is achieved solely by application process code which must be correct if crash tolerance is to work correctly.

The major difference between process survivability and crash tolerance as provided in ADNET and Tandem is that process survivability is transparent to the application programmer. We believe that this is the major advantage that process survivability has over these systems, as it ensures that crash tolerance is not vulnerable to application programmer error. That this approach is also taken by two recent systems - Auragen and DEMOS M/P - is encouraging, as it implies that transparency is important in a crash tolerant system, and that it valued by others.

5.4 Summary

Process survivability is a way of implementing an application process so that it can survive the crash of its host computer. Each process has a number of backups so that it can withstand multiple computer crashes even if they occur simultaneously. By implementing all of the control system's application processes in this way, the control system's software can survive a number of computer crashes and still fulfil its specifications.

Unlike ADNET and Tandem, process survivability is transparent to the application programmer. This ensures that programmer errors cannot prevent process survivability from being performed correctly. Transparency also makes process survivability suitable for adding to existing distributed computer control systems, as there would be no need to retrain programmers and it would not be necessary to alter existing software.

Moulding (1980a, page 9) states that using standby redundancy "introduces immense backward error recovery problems and in practice no such automatic reconfiguration has been achieved". Process survivability removes these "immense backward error recovery problems" by achieving consistency without recourse to 'dominoing'.

The rest of this thesis describes how process survivability could be implemented for the paper distributed computer control system PROSUR.

6. PROSUR and the Foundations of Process Survivability

6.0 Introduction

PROSUR (PROcess SURvivability) is the distributed computer control system that was designed as an environment in which to develop process survivability. PROSUR is a 'paper' distributed computer control system, that is, it exists on paper but it has never been implemented.

PROSUR conforms to the three-level model of a distributed computer control system described in Chapter 2, consisting of a hardware level, distributed kernel level and application level. Process survivability is provided by a fourth level interposed between the distributed kernel level and the application level.

This chapter describes PROSUR's three standard levels and introduces the implementation of the process survivability level. Then in the following three chapters the implementation of the process survivability level is developed.

To facilitate the development of process survivability, PROSUR has been kept simple. While it has not been simplified to the extent that the value of process survivability has been nullified, it has resulted in PROSUR having a number of limitations as a distributed computer control system. The final section of this chapter outlines these shortcomings.

6.1 Application level

The application level is organised as a static configuration of processes communicating with each other by asynchronous message passing.

A process is a named running instance of a program. All programs are written in PROSUR's P/L which has been adapted from Pascal (Jensen and Wirth 1978) by removing the file handling facilities (including READ and WRITE etc.) and by adding message passing and device driving commands.

Processes are connected together by uni-directional 1-to-1 typed channels, where a channel is a queue of messages. Each channel has only a single sender process and a single receiver process: there is no facility for many-to-1 or 1-to-many connections. The sender places messages into the channel using a SEND command and the receiver removes these messages in the order in which they were sent using a RECEIVE command. Messages that have been sent but which have not yet been received are buffered within the channel.

The application level is created at system initialisation time by commands issued to the system manager. Application processes are created, named and allocated to computers, and the pattern of interconnecting channels that bind these processes together is specified. PROSUR is a static system, and so the configuration specified at system initialisation time remains constant throughout the control system's life. If the control system has to be altered in any way, for example to add or replace a process, then it must be turned off and then recreated. The only exception to this is the migration of processes between computers as a result of the actions of process survivability after a crash. After system initialisation the system manager has no further role.

A channel is formed by linking together the sender's output channel and the receiver's input channel. Input channels and output channels are declared within the programs by statements of the form:

```
To   : OutputChannel [10] Of letter;

From : InputChannel [10] Of letter;
```

where 'letter' is the base type of the channel and 10 is the maximum number of messages that can be buffered within the channel. Only input channels and output channels of the same base type and size can be linked together to form a channel. This linking is performed by the system manager at system initialisation time.

'To' and 'From' are the local names of the output and input channels. All interprocess communication is with respect to the local names of a channel. Hence the physical location of the processes is hidden from the processes. Such addressing is said to be 'location independent', and it is a vital feature in a system that incorporates process survivability. With process survivability a computer crash results in application processes being 'moved about', and location independent addressing (assuming adequate support from the kernel) ensures that the application processes will still be able to communicate with each other despite the changes in their physical locations. (Without location independent addressing it would be necessary to recompile the processes in order to adapt them to the altered configuration.)

The SEND and RECEIVE commands are of the form:

SEND (to, data, status, timeout)

and

RECEIVE (from, data, status, timeout).

SEND places the message 'data' into the channel whose local name is 'to'. RECEIVE takes the first message in the channel 'from' and places it into 'data'. Channels are typed and so in both commands the channel's base type must be the same as that of the 'data' parameter.

If a process issues a RECEIVE on an empty channel then that process will be suspended until a message is available. If a process issues a SEND on a full channel, that is a channel in which all of the buffering is taken up by outstanding messages, then that process will be suspended until space is available in the channel.

The 'timeout' parameter in both commands allows a process to maintain its responsiveness. If a SEND/RECEIVE takes longer than the period specified by timeout, perhaps because of the above flow control reasons or in the case of a SEND because the network is busy, then the command is aborted with the 'status' parameter set to 'failed'. If a SEND/RECEIVE terminates normally 'status' is set to 'succeeded'.

PROSUR suffers from the generic problem associated with asynchronous message passing: if a SEND fails due to the timeout expiring it is possible that the message has in fact been successfully delivered. (The reason for this is explained in Section 6.3 of this chapter.) It is left to the application processes to cope with this problem, possibly by implementing some higher protocol amongst themselves.

The final interprocess communication command is the boolean function:

PENDING (from) : Boolean;

which returns true if there is a message in the input channel 'from', and false otherwise. PENDING allows a receiver to check whether there is a message ready to be received before actually issuing a RECEIVE.

An input/output device is controlled by a process known as its 'handler'. Other processes can only access a device indirectly by communicating with that device's handler by message passing. The device handlers provide the rest of the application level with a high level, message-based interface to their devices.

Like channels, devices are declared within their device handlers by declarations of the form:

```
lp : LINEPRINTER;
```

where the type can be any of the system defined device types. The handler refers to its device by the local name. The device that is actually controlled is not specified until system initialisation time.

Device handlers control their devices using the DOIO procedure, which is based on the IO command in Concurrent Pascal (Brinch Hansen 1977). DOIO is of the form:

```
DOIO (dev, oper, data, stat, arg).
```

DOIO causes the device 'dev' to perform the operation 'oper' (where oper can be INPUT, OUTPUT, MOVE or CONTROL), with any further information concerning the operation being supplied by 'arg'. In the case of an input/output operation the value input/output will be read from/placed into 'data'. The outcome of the operation, for example COMPLETED or INTERVENTION, is returned by the parameter 'stat'.

The system manager ensures that the configuration specified is correct, as error checking at system initialisation time ensures that expensive runtime error checking need not be performed. In particular it ensures that each device has only one handler and that both are on the same computer, and that each input/output channel is bound to only one output/input channel and that they are of the same size and type.

6.2 Process survivability level

Process survivability is based on standby redundancy of application processes. An application process and its redundant backup copies together constitute a process-set. Every application process (and that includes device handlers) is implemented as a process-set.

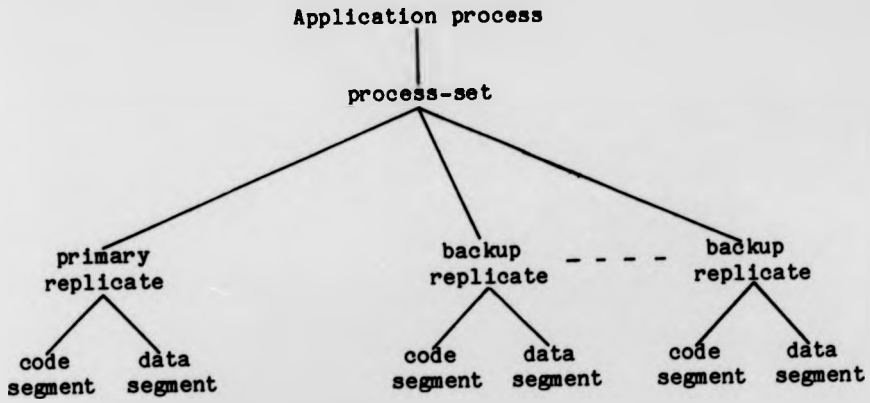
The process survivability level fulfils two roles:

1) It provides the process-set.

2) It ensures that any inconsistencies that arise between processes after a crash are removed.

In this section we describe how a process-set is organised and what it does, but the details of how a process-set is implemented are left until Chapter 9. Neither does this section describe how consistency is restored. The next two chapters describe the inconsistencies and explain how they are removed. The implementational details of restoring consistency are presented in Chapter 9.

A process-set consists of the primary replicate which is the running copy of the process, and of a set of non-running copies of the primary replicate called backup replicates (see Figure 6a below). Each replicate is hosted by a different computer. When the primary replicate is lost due to the crashing of its host computer one of its backup replicates is activated to replace it. The remaining backup replicates provide standby redundancy for this new generation of the application process.



The organisation of a process-set

Figure 6a

A primary replicate is the running copy of the application process. It consists of a code segment and a data segment. The code segment contains the compiled PROSUR's P/L program that defines the process, and the data segment contains the process's global variables, runtime stack and heap storage.

Each backup replicate also consists of two segments: the code segment, which is a copy of the primary replicate's code segment, and the data segment, which is a copy of the primary replicate's data segment.

A primary replicate's code segment is constant and so copies of it can be placed into its backup replicates' code segments at system initialisation time. The primary replicate's data segment alters all the time and so the backup replicates' data segments must be updated as the primary replicate runs. It would be impossible for the backup replicates' data segments to be continuously updated. Instead they are updated at intervals when the primary replicate performs a secure point. A secure point results in secure point data being sent to each of the backup

replicates. Secure point data contains sufficient information for each backup replicate's data segment to be updated so that it is an exact copy of the primary replicate's data segment.

Secure point placement is decided upon at runtime. There is no coordination of secure pointing between primary replicates. The insertion algorithm is a fundamental part of the implementation of process survivability. The development of the insertion algorithm and its implementation are described in Chapters 8 and 9.

When a primary replicate is lost one of its backup replicates is activated to replace it. The 'reincarnated' primary replicate will start to execute its code from the instruction immediately after the last secure point that was successfully executed by the previous generation of primary replicate.

A process-set's backup replicates are ordered linearly. This ordering defines the sequence in which the backup replicates will be activated in the event of the primary replicate being lost. A backup replicate's predecessors consist of the current primary replicate and of those backup replicates that are before it in the sequence. When a backup replicate's predecessors have all been lost it will be activated to become the new primary replicate. For convenience, we refer to the first backup replicate in the sequence as the senior backup replicate and the others as the junior backup replicates.

When a computer crashes a number of primary replicates are lost and their senior backup replicates will be activated to replace them. At the same time a number of junior and senior backup replicates are lost as well. The order of backup replicate activation is resilient to the loss of backup replicates: as a backup replicate is simply activated when its predecessors have all been lost, it does not matter that some of them were lost before they were activated.

Process survivability provides 'n out of m' crash tolerance: the control system can withstand the loss of n of its m computers. As we shall explain in Chapter 8, application processes play an active role in recovering each other after a crash. For a control system to withstand n crashes it is necessary that every application process has n backup replicates as otherwise processes would be lost and would not be able to assist each others' recovery.

The position of the primary replicates is specified to the system manager at system initialisation time along with the level of redundancy required for all application processes. The position of the backup replicates is decided upon by the system manager. The system manager also initialises the backup replicates' code and data segments.

PROSUR is a static system, and the configuration cannot be altered (other than by process survivability) while it is on-line. The only way that a crashed computer can be reintegrated into the system is by re-initialising the entire control system. For the same reason, when a primary or backup replicate is lost in a crash there is no attempt to maintain the level of redundancy by generating another replicate. The system has an initial level of redundancy. Once that many computers have crashed the redundancy for some processes will have been exhausted, and any further crashes will cause the control system to fail.

We have adopted the same approach to device redundancy as used in ADNET where the "failure of a peripheral computer will result in the loss of that peripheral from the system" (Moulding 1980b, page 8). Devices are single-homed: each device is only attached to a single computer. The result of this is that when a computer crashes any devices that are connected to it can no longer be used.

Although the devices in PROSUR are single-homed the device handlers are still implemented as a process-set. The original primary replicate is hosted by the computer to which the device is connected. The backup replicates will be hosted by computers that are not attached to the device, and so when a backup replicate is activated it will not be able to control its device. Device handlers are implemented as process-sets for two reasons:

1) As we shall explain later, processes assist in each others' recovery, and so if a device handler were lost it could not assist in the recovery of other processes.

2) The backup replicates can provide an intelligent response to further requests for services.

Does the loss of control over the devices nullify the advantages of process survivability? Obviously the designers of ADNET do not think so, but the opposite view is taken by the designers of HXDP who think that "processors in distributed real-time control systems are typically located near the sensors and actuators they serve; reconfiguration is rendered ineffective by the inability to move the function of the external devices" (Boebert et al. 1978, page 255). This difference in view is probably due to the fact that HXDP tends towards a process to computer ratio of 1:1, whereas ADNET and PROSUR have a much higher ratio and so computers are not concerned solely with controlling devices.

If a particular device is vital then it will be replicated, and each replicate will be connected to a seperate computer. Such an arrangement would be necessary anyway, as otherwise the control system would be extremely vulnerable to device failure. Each device will have its own device handler. In the event of a device handler crashing, its activated backup replicate would advise its users that it can no longer

control its device, and then the users could use one of the alternative device handlers instead. (Unfortunately the absence of dynamic channel linking in PROSUR makes this inelegant as it requires a process to be permanently connected to each of the alternative device handlers.)

Farber (1978, page 387) states that "peripherals have to be connected not only to one but to several processors". Then if one computer crashes one of the other computers could control the device instead, thereby maintaining the service.

Unfortunately the connection between the computer and the device is vulnerable to failure. Were multi-homing to be adopted in PROSUR then it would be possible for the primary replicate's host to become disconected from the device, leading to the situation where the primary replicate can no longer control its device but its backup replicates could.

The Tandem 16 uses multi-homed disks. Tandem is not troubled by the above problem as it is possible for a primary replicate and its single backup replicate (processes in Tandem have only one backup) to swop roles, thereby ensuring that the primary replicate is always controlling the device. In PROSUR the primary replicate cannot swop roles with one of its backups.

A possible solution to this problem would be to replicate the computer-device connection, thereby making it effectively invulnerable. However in a widely dispersed configuration this would lead to a prohibitively large amount of wiring. In Chapter 8, Section 8.3, we suggest a way, based on the Tandem approach, in which multi-homed devices and process-sets can be integrated so that a restarted backup replicate can control its devices.

To summarise, devices are single-homed and their device handlers are replicated. In the event of either the device failing, or the computer it is attached to crashing, the service provided will be lost. If a redundant I/O service is required then it would have to be implemented within the application level.

6.3 Distributed kernel level

Each computer runs a functionally identical kernel. These kernels together constitute the distributed kernel level. The distributed kernel level, along with the hardware level below it, provides the services that are used to implement the application level and the process survivability level.

In general the kernels operate independently of each other. Each kernel supports its local processes by providing time-sharing, virtual memory management, asynchronous message passing and low-level device control. The kernels cooperate with each other to transfer messages from one computer to another.

Each primary replicate hosted by a kernel is described within that kernel by a data structure known as its 'descriptor'. A primary replicate's descriptor contains its status ('runnable', 'unrunnable'), a copy of its volatile environment and its page table.

Primary replicates on the same computer are time-shared by their host kernel. The descriptors of those primary replicates that are runnable are organised into the ready-queue, from which the kernel chooses the next primary replicate to be run.

While a primary replicate is being run it will make use of a number of registers such as the program counter. The values contained in these registers are the primary replicate's volatile environment. When a primary

replicate relinquishes the processor, either because it has become unrunnable or because it has been pre-empted, its volatile environment is stored in its descriptor. When the primary replicate is next chosen to be run its stored volatile environment is reloaded into the registers and the primary replicate continues where it left off. If the volatile environment was not saved it would be overwritten by the next primary replicate to be executed.

Virtual memory is in the form of paging. Program addresses consist of a page number and a word number pair. The physical location of a particular page in main memory or on backing store is defined by the primary replicate's page table which is stored in its descriptor.

The distributed kernel level provides a reliable asynchronous message passing service that is used by the process survivability level to implement the interprocess communication mechanism used by the application processes. This service, which is very similar to the application level's communication mechanism, comprises 1-to-1 buffered channels for linking primary replicates together and a suite of kernel procedures for using them.

These channels are called pipes so as to distinguish them from interprocess channels. A pipe consists of an input pipe and an output pipe, which are linked together at initialisation time when the interprocess channels which they implement are linked together. Input pipes, like input channels, have a type and a size, but output pipes only have a type. The buffering needed for a pipe is provided by the receiver's kernel and forms part of the receiver's descriptor. In effect outstanding messages are buffered within the input pipe and the input pipe's size is the number of messages that can be buffered within it.

The kernel procedures for supporting interprocess communication are:

a) ksend (op: output pipe; m: message; Var s: status; t: timeout)

The message 'm' is sent down the output pipe 'op'. If the timeout period 't' expires or if the receiving primary replicate crashes then the ksend will fail and 's' will be set to 'failed'; otherwise 's' is set to 'succeeded'.

b) kread (ip: input pipe; Var m: message; Var s: status; t: timeout)

A message is copied from the input pipe 'ip' into 'm'. The message is not removed from the input pipe but it cannot be re-kread. If a message does not exist the caller will be delayed until a message arrives or until the timeout period 't' expires. If the kread was successful 's' is set to 'succeeded'; otherwise it is set to 'failed'.

c) kfree (ip: input pipe)

Kfree removes from 'ip' the messages that have been kread in since the last call of kfree. The space they occupied can now be used again.

d) kreceive (ip: input pipe; Var m: message; Var s: status; t: timeout)

This is identical to kread except that the message is removed rather than copied.

e) kpeeklast (ip: input pipe; Var m: message; Var s: status; t: timeout)

This returns a copy of the newest message in 'ip'. The message is not removed from the channel and it can still be input by a kread or a kreceive. The space occupied is not released by a kfree.

In all of these kernel procedures a timeout of minus one (-1) means for ever, and a zero (0) timeout means that the command will only succeed if it can be performed successfully immediately. These two values can only

be used sensibly with certain of these procedures and then only under certain circumstances.

Within the distributed kernel level all of the computers, application processes and pipes are identified by numbers. Computers and application processes are uniquely numbered, although all replicates within a process-set have the same identity. Input and output pipes are uniquely numbered within their host replicate.

Channel addressing within the application level is by local name, and similarly pipes are addressed by local input pipe and output pipe numbers. In order for a kernel to deliver a message sent by a local process it must know the address of the recipient input pipe. An input pipe's address has three components: process number, input pipe number, and the number of the computer that hosts the receiver's primary replicate.

The computer component of an input pipe's address will alter as the application process moves around the control system as a result of crashes. Because of this, the mapping from output pipe number to input pipe address is done at the time the message is sent and not during compilation or at system initialisation time. The mapping is defined by a table contained within the sender's descriptor. For every output pipe number the table contains the recipient input pipe's address. For example, in Figure 6b below, the primary replicate's first output pipe is linked to the first input pipe of process number 6, whose primary replicate is resident on computer number 2. When a primary replicate calls ksend the kernel uses the 'op' parameter to look up in the caller's descriptor the address of the recipient input channel.

		input pipe address		
		computer	process	input pipe
	1	2	6	1
output pipe	2	3	3	9
number	3			
	.			
	.			
	.			
	n			

Figure 6b

All of the figures needed to fill these tables are provided by the system manager at initialisation time. After a crash it is necessary for some of these mapping tables to be altered to reflect the current location of the application processes. This is done by the affected application processes themselves using the kernel procedure:

kconnect (op: output pipe; c: computer number);

which tells the kernel to alter the caller's mapping table to reflect the fact that the receiver connected to the caller by output pipe 'op' is now on computer 'c'. How kconnect is used is described in Chapter 9, Sections 9.3.2 and 9.3.4.

Any ksend performed on an output pipe that needs to be reconnected, but which has not yet been reconnected, will fail as the recipient computer specified in the mapping table no longer exists.

A reliable interprocess communication mechanism is one in which messages are neither lost nor corrupted. To achieve this the kernels employ a fault tolerant protocol such as Byte Stream protocol (Johnson 1980) to transfer messages between computers.

Byte Stream protocol provides reliable virtual circuits. It uses checksums and positive acknowledgements to detect errors, and uses repetition after a timeout period to recover from the loss. In order that this recovery does not result in messages being replicated, both messages and acknowledgements carry a sequence number. These virtual circuits are created at system initialisation time, and they are reset in order to re-initialise the sequence numbers after a crash of either the sender or the receiver.

Unfortunately such protocols suffer from a generic problem. If the timeout specified by the sender expires before an acknowledgement is successfully received then the ksend will return with its status parameter set to failed, but the message may have been delivered successfully and the acknowledgement lost. As mentioned in Section 6.1, coping with this problem is left to the application processes.

There is a limit to the size of message that can be sent at one time. If the message specified in a ksend is larger than this maximum size, then it is split up and the constituent parts are sent separately. When all of the parts have been received by the receiver's kernel the message is reconstituted and placed into the input pipe. All of this is transparent to the application level.

The kernels provide a procedure for controlling devices called kdoio which is used by the process survivability level to implement DOIO. Kdoio has the same parameters as DOIO except that the device is identified by a number rather than a variable name.

Every device attached to a kernel's host computer is described within the kernel by a data structure called its descriptor. A device's descriptor contains the information concerning the device that is needed to implement a call of kdoio on that device. A primary replicate's descriptor contains a table of pointers to the device descriptors of the devices it

handles. The device's identity specified in a call of kdoio is the index to that device's descriptor contained within this table.

Devices have only one handler and so there is no complicated sharing. The possible exceptions to this rule are those disks which are partly used for paging and which are also controlled by a device handler. In this case the disk would be controlled by an executive process that handles requests from both the paging system and the disk's handler.

A backup replicate is organised in exactly the same way as a primary replicate. It has a descriptor and it is allocated pages for its data and code segments. The descriptor includes the buffering for the backup replicate's input pipes; even though the backup replicate may never be used this buffering must be available in case it is. If there is not sufficient storage in the kernel for the backup replicates' input pipes then either the level of redundancy is too high or the number of computers is too low. Minor differences of a backup replicate from a primary replicate are:

1) its status is 'unrunnable';

2) in its output-pipe mapping table the computer address column is full of zeros;

3) the table of pointers to device descriptors is empty as the devices are not attached to the backup replicate's host computer;

4) all of its pages are held on backing store.

Although we have described the services provided by a kernel as if it were a single entity, it would be implemented as a 'core-resident' nucleus and a number of executive processes. The executive processes would be handled by the nucleus in the same way as a primary replicate is. To call a kernel procedure such as ksend, the primary replicate places the

parameters in a location known to the kernel and then raises a software interrupt.

For PROSUR to support process survivability the distributed kernel level must be able to continue normally despite the loss of one or more of the kernels. To do this none of the kernels must contain data or provide services that are needed by any of the other kernels. This is the case in PROSUR, as the kernels have their own copies of the necessary tables and none of them exports a service to the others. Also, if while a message is being transferred between computers, one of the kernels crashes, the other must be able to abort the transfer without waiting for the user specified timeout to expire.

There are two characteristics of the message passing system that are needed for process survivability. One of these, the late addressing of messages by the kernel, has already been discussed. The second is reliability.

Secure point data is transferred between computers using the message passing system. Were secure point data to be corrupted during transmission and were that corruption not noticed, then the backup replicate that is updated using that corrupted secure point data would itself be corrupted. Were the corrupted backup replicate to be activated it would fail, and this could cause the control system to fail as well.

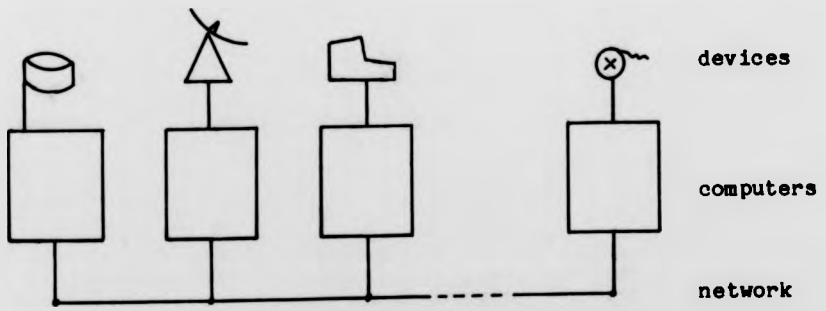
Fortunately, even if the message passing service is less than 100% reliable, the probability of a corrupted backup being activated is still very small. As we shall describe later, when a secure point is executed the secure point data is sent to each backup replicate in turn, and also it is sent to each of them as a sequence of smaller messages. Hence, any corruption will most likely be limited to a small part of a single backup replicate. Furthermore, prior to a corrupted backup replicate being activated, the corrupted part may have been overwritten and corrected as a

result of a later secure point.

The network overheads involved in providing a 100% reliable message passing service are likely to be very high because of the level of redundant data that would have to be transmitted. A less than 100% reliable service would be more practicable, and as long as the probability of a corrupted backup replicate being activated is sufficiently small then it would be an acceptable economy. Whether or not a 100% reliable service could ever be provided is an arguable point, and so the risk of corruption is probably unavoidable anyway.

6.4 Hardware level

PROSUR's hardware, which is illustrated in Figure 6c, consists of a number of identical computers joined together by a local area network. Each computer has sufficient local backing store to support virtual memory requirements.



PROSUR's hardware

Figure 6c

Each input/output device is attached to a single computer. This computer will host the device handler's original primary replicate. The device is not attached to those computers that host the device handler's backup replicates. In certain situations some of the computers will be on-board computers, that is, they will be an integral part of the devices they control.

If the computers were not homogeneous, messages and secure point data originating from one computer would have to be translated before they could be used by another type of computer. This would necessitate the use of an intermediate format for data being sent between computers, as is the case for parameter and message passing in DLCN (Liu and Reames 1977). Furthermore, "the clarity that homogeneity provides in allowing one to see a single research problem at a time is very appealing" (Saltzer 1978, page 49).

The computers are physically dispersed in order to reduce the chances of multiple computer crashes. The distances involved would depend on the perceived threats to the system. Even in a civilian application some level of dispersal would be desirable as a protection against such events as fires and floods.

Were the network to be broken then, depending on the network's type, the control system would by partitioned into two or more isolated groups of computers. Each group would consider themselves to be the only surviving computers and backup replicates would be activated in order to recover from the loss of those computers not in the partition. Partitioning leads to a number of problems:

a) some partitions would be too small to support the full control system;

b) the separate partitions, whether complete or not, would compete with each other to fulfil the overall task;

c) when the network is repaired it would be necessary to close down the control system and reinitialise it in order to remove duplicate copies of processes.

Tandem (Katzman 1978) avoids this problem by having a duplicated network and by only guaranteeing the system to tolerate a single hardware failure, and so by definition Tandem need not be concerned with partitioning. The network used in ADNET incorporates three or more redundant highways (Tillman et al. 1981) connected to the computers by spurs, and it is believed that in practice ADNET cannot be broken even in the hostile environment of a warship in action (Lakin 1982).

To avoid partitioning in PROSUR it is assumed that the network incorporates a high level of redundancy, and we amend our claims for process survivability to include the proviso that PROSUR will survive the failure of n computers (where n is the level of process redundancy) or the exhaustion of the network's redundancy.

In a commercial environment where the threat of physical damage is lower a network such as Planet (Gee 1983) which is basically a duplicated Cambridge Ring would provide sufficient redundancy. Interestingly, a military version of Planet has recently been developed for use on warships (Computing 1983) and it is probable that the level of redundancy in this version is even higher.

6.5 PROSUR's shortcomings

PROSUR has been deliberately kept simple so as not to hinder or divert the development of process survivability. This simplification means that PROSUR is not a 100% practical distributed computer control system, but it does form a good basis for one. In this sub-section we outline those facilities that have been removed in the cause of simplicity.

There are two major omissions from the programming language PROSUR's P/L. Firstly, in Chapter 2 we explained the need for a process to be able to wait for a message from any one of a group of channels, but in PROSUR's P/L this can only be accomplished by the unsatisfactory use of a busy loop incorporating a lot of PENDING calls. The second omission is the group of commands that enable a process to suspend itself for a period of time or until a specific time.

PROSUR only supports 1-to-1 linking of input channels to output channels. A practical system would also support many-to-1 and 1-to-many linkages.

PROSUR is a static system. In order to alter the application level it is necessary to re-create the system. Although there are process control applications where this would be acceptable, the majority of applications require a dynamic system so that the application level can be altered without having to turn off the control system.

Shortcomings in process survivability itself are discussed in the concluding chapter.

6.6 Summary

In this chapter we have outlined the organisation of the four levels of PROSUR, and we have described in general terms the way in which process redundancy is implemented. In the following two chapters we describe the inconsistencies that arise after a crash (these are similar to those already described in Chapter 4) and how they can be removed. Then in Chapter 9 we present the implementational details of the distributed kernel level and of the process survivability level in order to describe how process survivability can be implemented in PROSUR.

7. The Inconsistencies That Can Arise After A Crash

7.0 Introduction

In this chapter and in subsequent chapters, application processes are simply referred to as processes. Where this might lead to confusion with executive processes they are given their full title.

A process's state is defined by the values contained in its primary replicate's data segment and volatile environment. A process's state records its past actions in that the value of its state is the result of those past actions. When a process's primary replicate performs a secure point its backup replicates are updated so that their data segments and volatile environment are exact copies of those of the primary replicate.

Each channel links two processes together: the sender and the receiver. A channel is created at system initialisation time by linking together an output channel in the sender and an input channel in the receiver. Those outstanding messages that have been sent but which have not yet been received from a channel are buffered within the input channel half of that channel. A channel's state is defined by the messages currently buffered within its input channel half.

The controlled environment, which is the real world outside the I/O devices, also has a state. It is sufficient for our purposes to define the value of the environment's state as being solely the result of the control system's past actions.

The sequence of instructions that are executed by a process's primary replicate between two consecutive secure points is called a 'task'. The task that was being executed by a process's primary replicate when its host computer crashed is, for descriptive purposes, referred to as the process's 'interrupted-task'. (The primary replicate of course does not

know when it is executing an interrupted-task.)

When a computer crashes all of the primary replicates that were running on it are lost. The process-set mechanism activates the senior backup replicates of these primary replicates in order to replace them. In the following descriptions we will refer to a process's primary replicate simply as the process, and refer to the process's activated senior backup replicate as the restarted process.

When a process is restarted it will start to execute its code from the instruction immediately after the last secure point that has been completed prior to the crash, and its state will be the same as it was when that secure point was performed. The process loses all knowledge of the actions that it performed during its interrupted-task.

As was explained in Section 6.3 of the previous chapter, all of the outstanding messages in a channel are buffered within the channel's input channel half. When a process is restarted it has the necessary buffering for its input channels, but the pre-crash contents will have been lost.

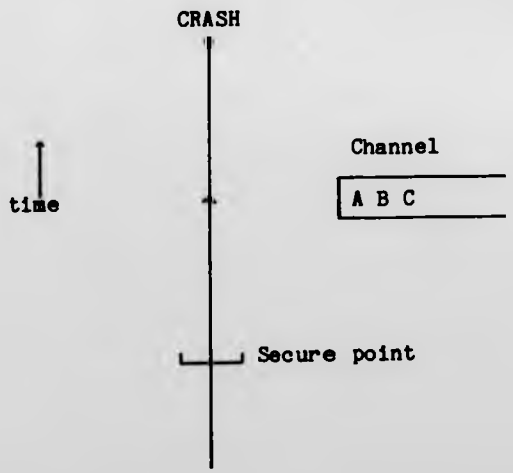


Figure 7a

As an example of the effects of a crash, Figure 7a above shows the execution of a process prior to a crash. It has established a secure point (represented by a horizontal '[' character) and its single input channel contains three outstanding messages (A, B and C).

After the crash, messages A, B and C will have been lost and the application process's state will have been reset to that which existed when the last secure point was established. The restarted process will start to execute its code from that secure point.

The resetting of processes' states and the loss of outstanding messages leads to a number of inconsistencies arising between the processes' states, the channels' states and the controlled environment's state.

If all of the commands executed by a process simply manipulate its own state then there will be no relationship between that process's state and the channels' states, the control environment's state, and the other processes' states. When such a process is restarted there can be no inconsistencies.

Inconsistencies arise when a process uses those commands that either affect states other than its own, or return values that are based on states other than its own. In PROSUR's P/L these commands are SEND, RECEIVE, PENDING and DOIO.

This chapter describes the inconsistencies that can arise after a crash, and it explains the combinations of circumstances that cause each of them to arise.

The first section of this chapter describes the inconsistencies that arise when two processes are linked together by a single channel and the host of one or both of the processes crashes. The second section describes the inconsistencies that arise between the state of a device

handler (with a single device) and the state of the controlled environment when the device handler's computer crashes.

A process may have several channels connecting it to several other different processes, and a device handler may control one or more devices as well as having several channels. In the final section of this chapter we describe the cumulative effects of the inconsistencies that arise on every channel and every device.

7.1 Inconsistencies between two communicating processes

7.1.0 Introduction

A channel connects a sender to a receiver. In a crash the sender may be restarted, or the receiver may be restarted and the channel contents lost, or both of these may occur. The following three sub-sections describe the inconsistencies that can arise in each of these three cases.

7.1.1 Sender restarted

The sender and receiver processes are on different computers, and the sender's host computer crashes, but the receiver's host does not. As a result of this:

a) The sender is restarted from its last successfully completed secure point, that is from the start of its interrupted-task. (If a secure point was being performed at the time of the crash then that secure point will be discarded and the previous one used.)

b) The receiver process is unaffected and it continues to execute its code uninterrupted by the crash.

c) The channel is also unaffected as its outstanding messages are buffered within its input channel end which is stored on the receiver's computer and that computer did not crash. None of the outstanding messages in that channel are lost.

Were the sender to have sent any messages during its interrupted-task then these messages will have survived the crash, either within the channel if they are still outstanding, or as part of the receiver's state if they have been RECEIVEd. (If a message was being sent at the time of the crash then any part of that message that had been received by the receiver's kernel will be discarded - messages are either received completely or not at all.) However, the sender has been reset to the state that it was in at the start of its interrupted-task (which is before those messages were sent), and that state does not record that these messages have been sent. With respect to the sender's current state those messages that were sent by its interrupted-task have not yet been sent, but they already exist.

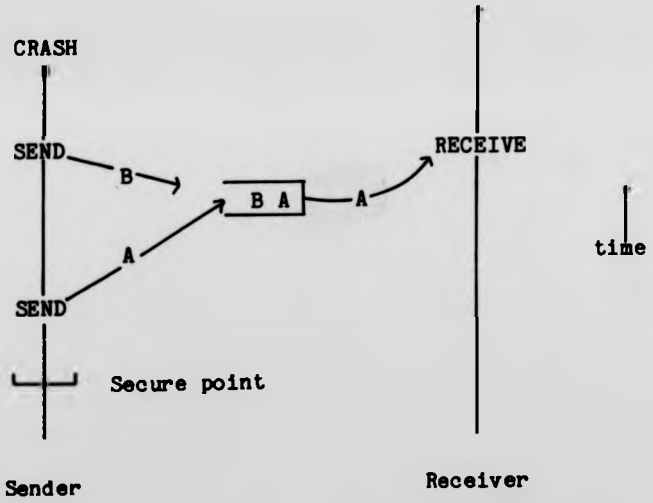


Figure 7b

An example is shown above in Figure 7b of two processes at the time of a crash. Prior to the crash two messages (A and B) were sent by the sender during its interrupted-task. After the crash the sender will be restarted from its last secure point and the two messages will survive, which is an inconsistency.

7.1.2 Receiver restarted and channel contents lost

The sender and the receiver are on different computers, and the receiver's host computer crashes but the sender's host does not. As a result of this:

a) The receiver is restarted from its last successfully completed secure point, that is at the start of its interrupted-task.

b) Any outstanding messages within the channel will be lost as they were buffered within the channel's input channel end which was stored on the receiver's computer.

c) The sender is unaffected and it continues to execute its code uninterrupted by the crash.

Any message that was outstanding at the time of the crash, or which had been RECEIVEd during the receiver's interrupted-task, will be lost. The only messages to have survived the crash are those that were RECEIVEd prior to the receiver's last secure point. The sender's state however will record the sending of every message that it has sent, including those that were lost in the crash. With respect to the sender's current state those messages that it has sent and which were not RECEIVEd prior to the receiver's last secure point have been lost.

A further inconsistency can arise from using the PENDING function. If prior to its last secure point the receiver detects the presence of a message using PENDING but does not RECEIVE the message until after the secure point, then after the crash the receiver's state will still record that message's presence but the message will have been lost.

An example is shown below in Figure 7c of two processes at the time of a crash: messages A and B were sent by the sender; A was detected prior to the receiver's last secure point and then RECEIVEd after the secure point. After the crash the sender's state records that A and B have been sent, and the receiver's state records the presence of message A, but both A and B are lost.

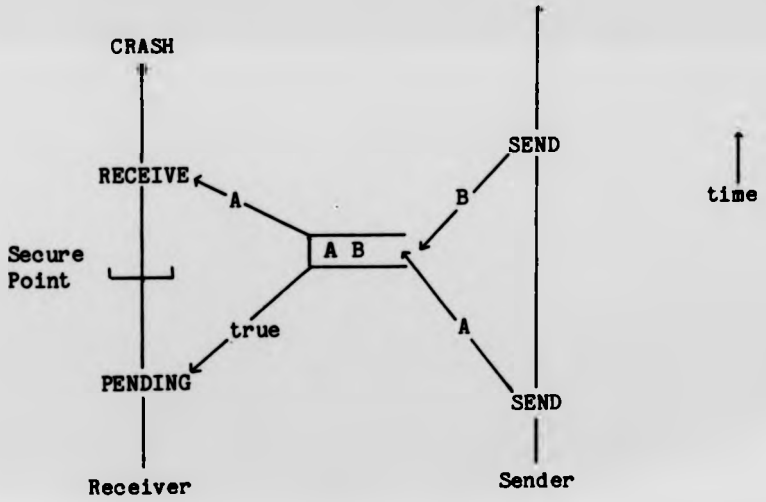


Figure 7c

7.1.3 Sender and receiver restarted, and channel contents lost

The sender and the receiver are on the same computer, and that computer crashes. As a result of this:

a) The sender is restarted from its last secure point.

b) Any outstanding messages within the channel are lost.

c) The receiver is restarted from its last secure point.

The sender's state will only record the sending of those messages that it sent prior to its interrupted-task, and the only messages to survive are those that were RECEIVEd by the receiver prior to its last secure point. The resultant inconsistencies will be one of the two cases described in the previous two sub-sections depending on whether, with respect to the sender's and receiver's current states, more messages have been sent than have survived or whether more messages have survived than have been sent.

Inconsistencies may also arise from the use of PENDING. Firstly, if a message's presence was detected prior to the receiver's last secure point (but the message was not RECEIVEd) then after the crash the record of the message's presence will survive but the message itself will have been lost. Secondly, were that message to have been sent after the sender's last secure point then no record of it being sent would survive the crash either. The receiver's state would record the presence of a non-existent message that has not been sent.

Any message that was sent after the sender's last secure point and which was neither received nor detected prior to the receiver's last secure point does not result in an inconsistency, as the message and all record of its sending have gone. (Message B in Figure 7d below is such a message.)

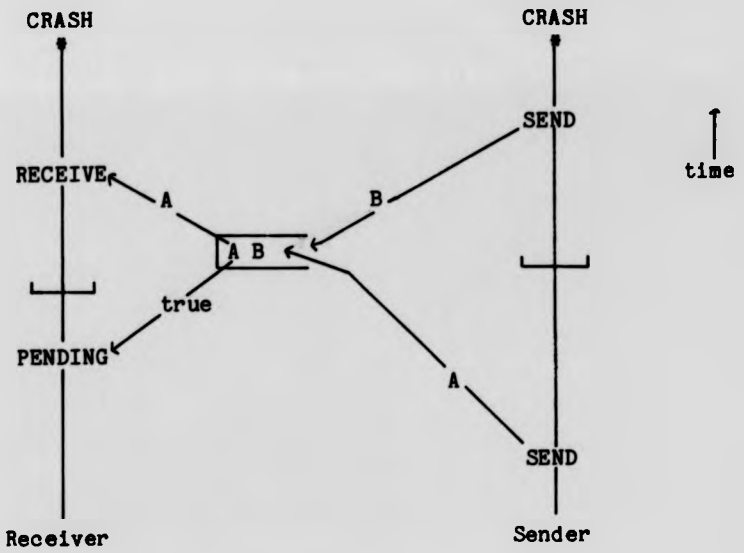


Figure 7d

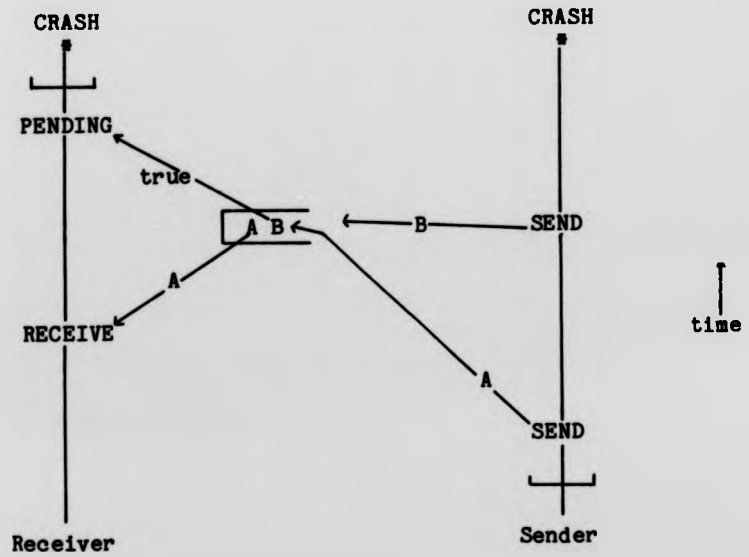


Figure 7e

In the first of the above two diagrams (Figures 7d and 7e) we illustrate the situation that results in more messages being sent than survive, and in the second we illustrate the situation that leads to more surviving than were sent. In both cases the inconsistency arising from the use of PENDING occurs. In the first diagram, the loss of message B does not lead to any inconsistency as after the crash it has neither been sent nor has it survived.

7.1.4 Summary

When a receiver crashes, all of the outstanding messages and all of the messages that it has RECEIVEd during its interrupted-task are lost. If a message is lost but the record of its sending survives then it is called a 'missing' message. It is the missing messages that cause the inconsistencies. Those messages that are lost but not missing do not cause inconsistencies. For example, in Figure 7d messages A and B are both lost but only A is missing.

When a sender crashes, it forgets that it has sent those messages that it sent during its interrupted-task. Of these messages, it is only those which survive the crash that cause inconsistencies, as they are messages that should not yet exist. These messages are called 'premature' messages. Those messages that are forgotten but not premature do not cause inconsistencies. For example, in Figure 7e messages A and B are forgotten, but only A is premature.

7.2 Inconsistencies between a device handler and the environment

When a device handler's computer crashes, the device handler will be restarted from its last successfully completed secure point. All knowledge of the actions performed by its interrupted-task is lost.

For our purposes the environment's state is determined by the past actions of the input/output devices and so when a computer crashes the environment's state is unaffected.

If the device handler had performed any DOIO commands during its interrupted-task then the environment's state will have been altered by them. However, as the device handler has been reset to a state that existed prior to the execution of these commands, its state will not record the performing of these commands. Also any results such as status and data values input by these DOIOs will have been lost.

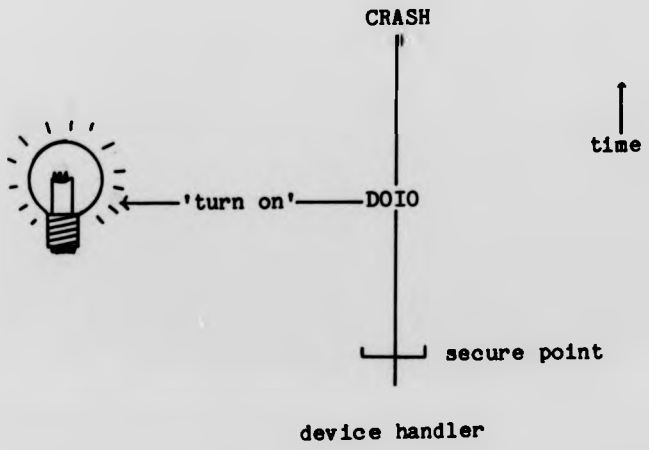


Figure 7f

Figure 7f above illustrates this problem. A device handler controls a light which it turns on during its interrupted-task. After the crash the light will still be on, but the device handler will have been

restarted from its last secure point and its state will not record that the light has been turned on.

These inconsistencies are similar to those that arise when only a sender is restarted (see Section 7.1.1) as the results of the DOIO (turning the light on) have survived but with respect to the device handler's current state the DOIO has not yet been performed. At the same time it is also similar to the case where only the receiver is restarted (see Section 7.1.2) as the value returned by the DOIO's status parameter has been lost.

The values output by DOIOs (or the actions they perform) during an interrupted-task are called premature output-data, and the values input by DOIOs during an interrupted-task are called missing input-data. A single DOIO may input and output at the same time.

7.3 Process-wide and system-wide inconsistencies

So far the inconsistencies that arise after a crash have been described on a per-channel and on a per-device basis. We now describe them on a per-process basis and then on a system-wide basis.

In general an application process will be linked by channels to a number of other application processes. Some of these channels will be declared within the application process to be input channels and the others will be declared to be output channels, and so an application process is both a sender and a receiver.

Figure 7g shows an application process as the hub of a circle of other application processes with which it communicates. The channels that connect the hub application process to its communicants are shown as lines. The lines are directed to indicate the direction of the message flow in the channels. (The communicants' other channels are omitted.)

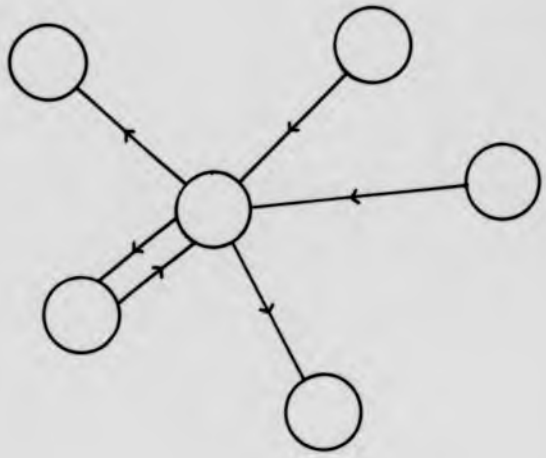


Figure 7g

These application processes may be located on one or more computers. A computer crash may result in one or more of the processes being restarted, and in the loss of messages from those processes' input channels. Henceforth a process that was on a crashed computer will be described as being crashed itself, implying that it has been restarted, and that the contents of its input channels have been lost.

We will first look at the inconsistencies that arise when the hub application process does not crash but all or some of its communicants do.

The overall inconsistencies that arise are simply the concatenation of the inconsistencies that occur relative to each channel. On every output channel where the receiver has crashed messages may be missing, and on the input channels where the sender has crashed there may be premature messages.

As an example Figure 7h below shows the hub application process H, communicating with two other application processes C1 and C2 prior to the crash of C1 and C2. After the crash message A will be premature and

messages B and C will be missing.

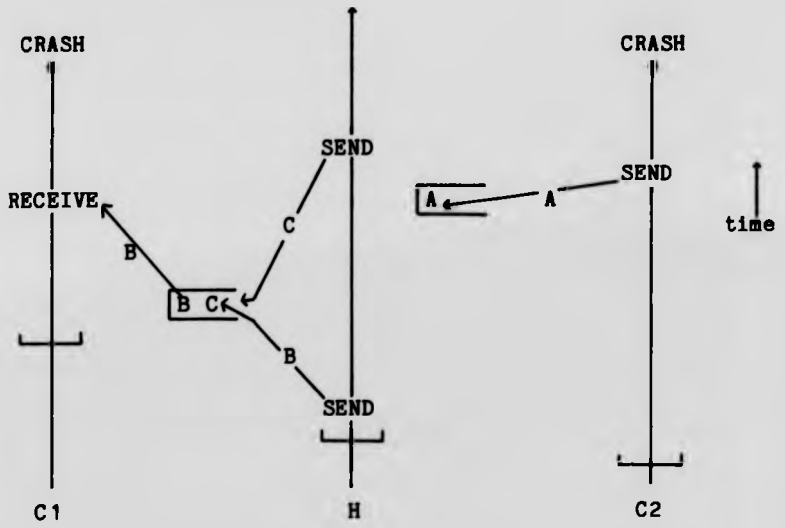


Figure 7h

If the hub application process crashed along with some of its communicants then again the overall inconsistencies would be the concatenation of the inconsistencies that arise on each channel. For example, if in Figure 7h process H alone crashed then message A would be missing, and messages B and C premature.

A device handler will have devices as well as channels. Again the overall inconsistencies that arise after a crash would be the concatenation of those arising on each channel along with those arising on each device.

The system-wide inconsistencies are simply the concatenation of the inconsistencies that arise with respect to each process.

7.4 Summary

In this chapter we have described the inconsistencies that may arise after a crash. One of the tasks of the process survivability level is to remove these inconsistencies after a crash. In the following chapter we outline how the process survivability level achieves this, and then in Chapter 9 full details of its implementation are given.

In this chapter we have only discussed the inconsistencies that arise after a single computer crash. We leave the discussion of multiple computer crashes until Chapter 9, as it is more sensible to discuss them after we have described how process survivability is implemented.

8. An Introduction to Consistency Restoration after a Crash

8.0 Introduction

Prior to a crash all of the processes will be in a consistent state - they will not be subject to any of the inconsistencies described in the previous chapter. After a crash a restarted process's state may be inconsistent due to:

1) Missing messages: those messages which, with respect to their senders' current states, have been sent to the restarted process but have been lost in the crash.

2) Premature messages: those messages that were sent by the restarted process's interrupted-task and which survived the crash.

3) A record of the presence of a message in an input channel (the result of a PENDING) survives the crash but the message whose presence is recorded is lost in the crash.

4) Missing input-data: the input-data that was input from the controlled environment by DOIOs executed during a device handler's interrupted-task and hence lost in the crash.

5) Premature output-data: the output-data contained in the controlled environment's state which was produced by DOIOs executed during a device handler's interrupted-task.

The restarted processes are recovered to consistent states by the process survivability level.

In this chapter we introduce the way in which the process survivability level performs process recovery, and we present arguments to show that this recovery achieves its aims. The implementational details of

process recovery are given in the second half of the next chapter.

As we shall explain in the first section of this chapter restarted processes are restored to a consistent state by the following:

1) The process survivability level causes the missing messages to be replaced by copies of the originals.

2) The PENDING related inconsistencies are prevented completely by implementing RECEIVE and PENDING so that detected messages are actually input as well.

3) Processes written in PROSUR's P/L are invariant: if a process, or part of a process, is repeated with the same input it will act in exactly the same way and produce exactly the same results every time. We shall argue that by restarting a process from its last secure point and by replacing any messages that are missing from that process's input channels, then that process will, by simply re-executing its interrupted-task, recover itself to a state that is consistent with any premature messages that it produced prior to the crash.

Unfortunately, time-dependencies in PROSUR's P/L prevent some of the inputs needed by a restarted process from being replaced in time. In order to ensure that these time-dependencies do not prevent correct recovery the process survivability level inserts a secure point prior to every SEND. Section 8.2 of this chapter describes these time-dependencies, explains how these can prevent a restarted process from recovering to a consistent state, and finally explains how, by inserting a secure point prior to every SEND, correct recovery can be achieved despite the time-dependencies.

The rest of the chapter is concerned with restoring consistency between device handlers and the controlled environment (which unfortunately cannot be achieved in PROSUR), and with the way that secure points must be executed.

8.1 Restoring consistency in an application process

8.1.0 Introduction

This section describes how the process survivability level recovers an ordinary application process (not a device driver) to a consistent state. Such a process is susceptible to the first three of the inconsistencies itemised in the introduction to this chapter.

As soon as a process is restarted it begins to execute its code; it is not held up while recovery is performed. Recovery is performed in parallel to the continued running of both the restarted processes and the unaffected processes (that is those processes that were not on a crashed computer).

Figure 8a below shows three processes at the time of a crash. Process-1 is unaffected by the crash, but Process-2 and Process-3 are both restarted from their last secure points. Process-1's input channel is not affected by the crash and continues to hold message D. The contents of Process-2's input channels are lost. This example will be used to illustrate the following descriptions.

8.1.1 Replacing the missing messages

Every time a process successfully SENDs a message, a copy of that message is stored within the sender's data segment. After a crash copies of the missing messages are re-sent so as to replace the missing messages. By storing the copies within the sender's data segment it is ensured that they will survive any crash as they will be backed up along with the rest

of the sender's data segment every time the sender performs a secure point.

Referring to Figure 8a, messages A and B are missing after the crash. (Message C is not classed as being missing as its sender has been restarted from a secure point prior to the SEND that produced message C.) Process-1's data segment contains a copy of message A and so this can be re-sent. Process-3 sent message B prior to its last secure point and so the copy of message B will have been backed up along with the rest of Process-3's data segment by Process-3's last secure point, and so message B can be re-sent.

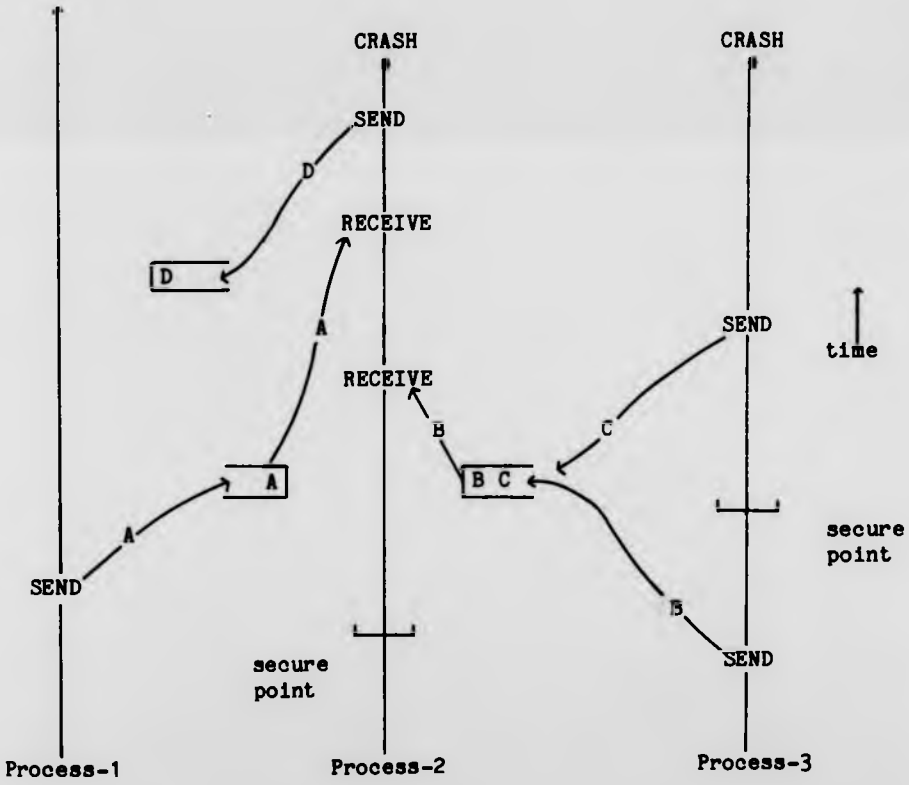


Figure 8a

The way in which the processes are stimulated to re-send the missing messages, and how they determine which messages are missing, is described in Chapter 9, Section 9.3.

8.1.2 Preventing the PENDING related inconsistencies

In Chapter 7, Section 7.1.2, we described an inconsistency that can arise from the use of the PENDING command. If prior to its last secure point a receiver detected the presence of a message using PENDING but did not RECEIVE that message, then when the process is restarted its state will record the message's presence but the message will be missing.

As such a message is a missing message it will be replaced by a copy. However, replacing messages is done in parallel to the restarted process's continued running. This means that if the restarted process tries to RECEIVE the detected message then the RECEIVE's timeout may expire before the message is replaced. This is itself an inconsistency as the process 'knows' that there is a message in its input channel and its RECEIVE should not fail.

This problem is completely removed by extending each input channel to include a single-message buffer which is stored as part of the receiver's data segment, and by implementing PENDING and RECEIVE in the following way:

a) PENDING (from : InputChannel) : Boolean

If both the input channel 'from' and its single-message buffer are empty then PENDING returns a value of false. If the buffer is empty and the input channel is not empty, then the first message in the input channel is placed into the buffer and PENDING returns a value of true. If the buffer is not empty then PENDING returns a value of true.

b) RECEIVE (from : InputChannel, ...)

If there is a message in from's buffer then that message is input and the buffer is emptied. If the buffer is already empty then a message is input in the normal way.

As the detected message is stored in the receiver's data segment at the same time as its presence is assimilated it is impossible for the record of a message's presence to survive a crash and not the message itself. This solution also removes the possibility of the PENDING related inconsistency described in Section 7.1.3 that arises when both the sender and the receiver of a channel crash.

Implementing PENDING and RECEIVE in this way is an example of the use of fault avoidance.

When a receiver crashes, any messages that were detected by PENDINGs prior to the receiver's last secure point will survive the crash even if the messages were not subsequently RECEIVEd prior to the receiver's last secure point. This must be taken into consideration when calculating which messages are missing and which are premature. We return to this again in Chapter 9, Section 9.3.

8.1.3 Premature messages

The restarted processes must be recovered so that their states are consistent with any premature messages. In our example, Process-2 and Process-3 must be restored to states that are consistent with the presence of message D.

Processes written in PROSUR's P/L are invariant. This means that if a process, or part of a process, is re-executed with the same input (data segment, volatile environment and messages) it will act in exactly the same way and will produce exactly the same results every time.

By ensuring that when a process is restarted it initially has the same data segment and volatile environment as it had at the start of its interrupted-task, and by ensuring that the restarted process RECEIVEs the same messages as it did during its interrupted-task, we guarantee that:

a) The restarted process will repeat those SENDs which, when executed during its interrupted-task, produced the premature messages. (In our example, this is Process-2's only SEND.)

b) Not only are those SENDs repeated, but they also produce exactly the same messages as they did when executed prior to the crash.

Hence a restarted process's state becomes consistent with both the presence of the premature messages and with the contents of those messages, and the existence of the premature messages is no longer an inconsistency.

Those instructions that form the interrupted-task are called the restarted-task when re-executed after a crash. By ensuring that a restarted process has the same input as its interrupted-task, we guarantee that the process's restarted-task will be executed, and that the restarted process will act in exactly the same way and produce exactly the same output as it did during its interrupted-task.

Executing the restarted-task includes re-executing all of the SENDs that were performed during the interrupted-task. If the result of such a SEND has survived the crash then the repeated SEND is called a late-SEND, but if the resultant message did not survive the crash then the repeated SEND is called a normal-SEND. In our example Process-3's second SEND is a normal-SEND as message C did not survive the crash, and Process-2's SEND is a late-SEND because message D did survive the crash.

If executed normally the late-SENDs would result in the premature messages being duplicated, which would introduce further inconsistencies. To avoid this the late-SENDs are 'gagged'. When a SEND is gagged, the message is not actually sent but the SEND's status parameter is set to 'succeeded'. Hence after a late-SEND the message exists (it survived the crash) and the sender 'knows' that the message exists (because the late-SEND returned a status of succeeded), and so the process's state is consistent with that premature message. By repeating and gagging all of its late-SENDs a restarted process is restored to a consistent state.

The normal-SENDs are executed normally. A description of how normal-SENDs and late-SENDs are distinguished is given in Chapter 9.

We now return to the example shown in Figure 8a. On restarting, Process-3 is consistent with its surviving messages as all of them were sent prior to its interrupted-task (the only message sent during its interrupted-task did not survive the crash). By ensuring that Process-2 RECEIVEs the same two messages (A and B) as its interrupted-task, it is guaranteed that Process-2 will repeat the SEND that originally produced message D. This late-SEND will be gagged so as to prevent D being duplicated. Process-2 will now be in a consistent state.

If a restarted process is to execute its restarted-task then its volatile environment and data segment must initially have the same value as they did at the start of the process's interrupted-task, and the restarted process must RECEIVE the same messages as it did during its interrupted-task. The implementation of the process-set ensures that the first of these requirements is met.

The messages that were RECEIVEd during a restarted process's interrupted-task will consist of:

a) Messages that were sent by unaffected processes (for example, message A in Figure 8a), and messages that were sent by restarted processes prior to their interrupted-task (for example, message B in Figure 8a).

b) Messages that were sent by restarted processes during their interrupted-tasks (for example, message C in Figure 8a).

The first group of messages, described in (a), is a sub-group of the missing messages. After a process is restarted all of its missing messages are replaced by copies of the originals. If during its interrupted-task a process RECEIVEs a message that will be missing after the crash (for example, both of Process-2's RECEIVEs in Figure 8a) then when that RECEIVE is repeated as part of the process's restarted-task it will input a copy of the message that it originally input. Hence, if a process's interrupted-task only RECEIVEd messages from group (a) (or if it did not RECEIVE any messages at all), then on restarting the process would execute its restarted-task and produce exactly the same messages as its interrupted-task. For example, in Figure 8a both Process-2 and Process-3 will act in exactly the same way as they did prior to the crash, as Process-3 does not input any messages at all, and Process-2 inputs two missing messages.

We come now to the second type of message that can be input during an interrupted-task: those messages that are sent to it by other restarted processes during their interrupted-tasks. If repeated RECEIVEs, that originally input a message from this group, can be shown to input the same message after a crash as before, then we will have shown that a restarted process will have exactly the same input as it did during its interrupted-task, and so will act in exactly the same way as it did during its interrupted-task. To show this, we present the following algorithmic argument (a worked example is given in Section 8.1.4) to show that the messages produced by a process's restarted-task are the same as those

produced by its interrupted-task.

1) As has been explained before, if a restarted process has only RECEIVEd replaced messages, or if it has not RECEIVEd any messages at all, then when it repeats a SEND that SEND will produce exactly the same message as when it was executed by the interrupted-task.

2) On restarting, each restarted process will execute until either it waits to RECEIVE a message that will be (or has been) sent by another restarted process, or until it reaches the end of its restarted-task. By (1) above, any SEND executed by these restarted processes prior to this will produce exactly the same message as it did when executed prior to the crash.

3) If a restarted process has finished its restarted-task then, by our previous arguments, the messages that it has produced will be the same as those it sent during its interrupted-task.

4) Of those application processes that issued a RECEIVE, one or more of them will RECEIVE a message from one of the other restarted processes.

One of these RECEIVEs must be satisfied unless the processes are dead-locked. Inserting secure points does not cause dead-locking, and so if the restarted processes are dead-locked it is because they were dead-locked prior to the crash and so they have been recovered.

5) The processes whose RECEIVEs were satisfied will continue to execute their code until they again either reach the end of their restarted-tasks or they wait to RECEIVE a message that will be sent by another restarted process. Any SEND re-executed by a restarted process since its previous RECEIVE will have produced the same message as it did during the interrupted-task as the restarted process's input has consisted of replaced messages and messages sent by other restarted processes which so far have been shown to be the same.

6) Continue from step 3 until all of the restarted processes have completed their restarted-tasks.

Hence every SEND performed by a process's restarted-task will produce the same message as it did when it was first executed during the process's interrupted-task. The late-SENDs will be gagged, and the normal-SENDs will be executed normally.

To summarise, in this sub-section we have shown that by restarting a process from its last secure point and by replacing the messages that are missing from its input channels, the restarted process will recover to a state that is consistent with its premature messages.

8.1.4 Summary and example

The way that PENDING and RECEIVE are implemented prevents the PENDING-related inconsistencies from ever arising.

Every time an application process SENDs a message the process survivability level stores a copy of that message in the sender's data segment. These copies are used to replace those messages that are missing after a crash.

After a crash the process survivability level replaces the missing messages by sending copies of the originals. The restarted processes, being invariant, will recover themselves to states that are consistent with their premature messages. Any late-SENDs are gagged by the process survivability level in order to prevent further inconsistencies from arising.

We now present a worked example of how consistency is restored after a crash. Figure 8b shows three processes at the time of a crash. Process-1 is unaffected by the crash and its channel will continue to hold message D. Process-2 and Process-3 are both restarted from their last

6) Continue from step 3 until all of the restarted processes have completed their restarted-tasks.

Hence every SEND performed by a process's restarted-task will produce the same message as it did when it was first executed during the process's interrupted-task. The late-SENDs will be gagged, and the normal-SENDs will be executed normally.

To summarise, in this sub-section we have shown that by restarting a process from its last secure point and by replacing the messages that are missing from its input channels, the restarted process will recover to a state that is consistent with its premature messages.

8.1.4 Summary and example

The way that PENDING and RECEIVE are implemented prevents the PENDING-related inconsistencies from ever arising.

Every time an application process SENDs a message the process survivability level stores a copy of that message in the sender's data segment. These copies are used to replace those messages that are missing after a crash.

After a crash the process survivability level replaces the missing messages by sending copies of the originals. The restarted processes, being invariant, will recover themselves to states that are consistent with their premature messages. Any late-SENDs are gagged by the process survivability level in order to prevent further inconsistencies from arising.

We now present a worked example of how consistency is restored after a crash. Figure 8b shows three processes at the time of a crash. Process-1 is unaffected by the crash and its channel will continue to hold message D. Process-2 and Process-3 are both restarted from their last

secure points and messages A, C and B will be lost from their input channels. Message D is a premature message.

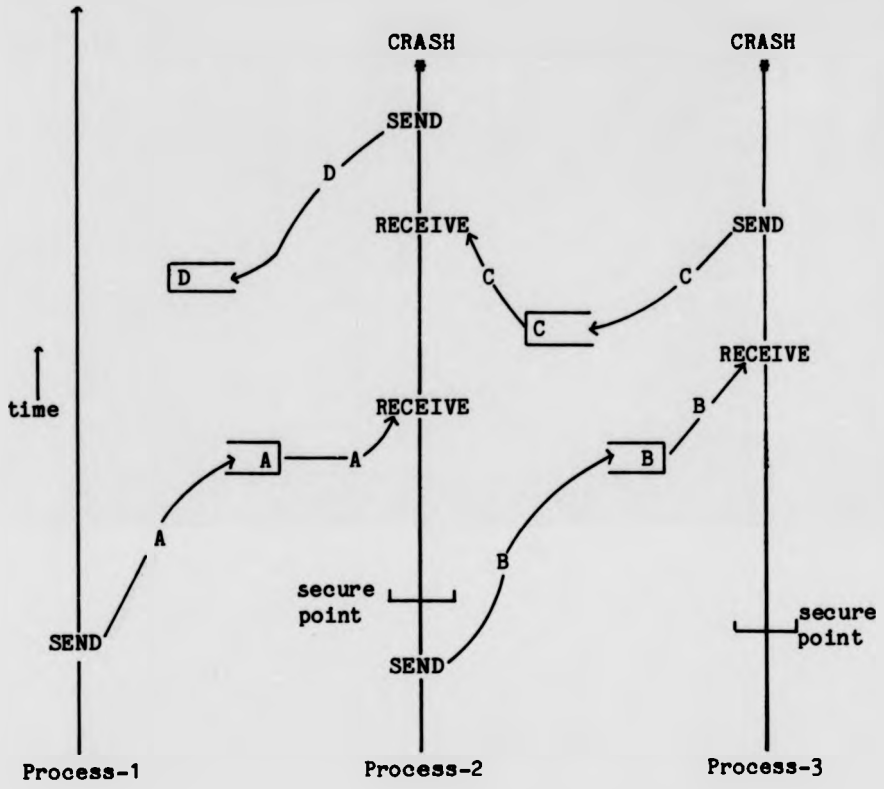


Figure 8b

Recovery would proceed in the following way:

1) A and B are missing messages and so will be replaced. We will distinguish between those messages that exist before and after the crash by appending suffixes of i and r respectively.

   We can state that Ai=Ar and Bi=Br, as they are copies of each other.

2) Process-1 continues to run normally.

3) Process-2 RECEIVEs the replaced message Ar. Process-2 then executes another RECEIVE, and as that input channel is empty it waits for Process-3's restarted-task to SEND a message.

4) Process-3 RECEIVEs the replaced message Br, and then SENDs the message Cr to Process-2. Process-3 then completes its restarted-task.

   Process-3's input consists of the message Br, and as Br=Bi - from (1) above - we can state that Cr=Ci.

5) Process-2's RECEIVE will terminate successfully, and Cr will be input. Process-2 will then perform the late-SEND which is gagged.

   Process-2's input consists of messages Ar and Cr. From (1) we know that Ar=Ai, and from (4) we know that Cr=Ci, and so Dr=Di, which is what we need for Process-2 to become consistent with its premature message Di.

6) All of the restarted application processes have finished their restarted-tasks and they are all consistent again.

8.2 Time-dependent functions and secure point insertion

8.2.0 Introduction

Figure 8c shows two processes at the time of a crash. During its interrupted-task Process-1 had detected the presence of message A and because of that it has sent message B to Process-2. Due to the crash Process-1 is restarted from its last secure point and the contents of its single input channel are lost. Process-2 is unaffected. After the crash message A is missing and message B is premature.

Message A will be replaced by the process survivability level which will cause the message to be re-sent. However the message may not have arrived by the time the PENDING is re-executed. If it has not arrived, then the PENDING will return a value of false and Process-1 will not re-execute its SEND. Because the PENDING returned a different value Process-1 has not recovered itself to a consistent state as message B exists but Process-1 does not know that it has sent it.

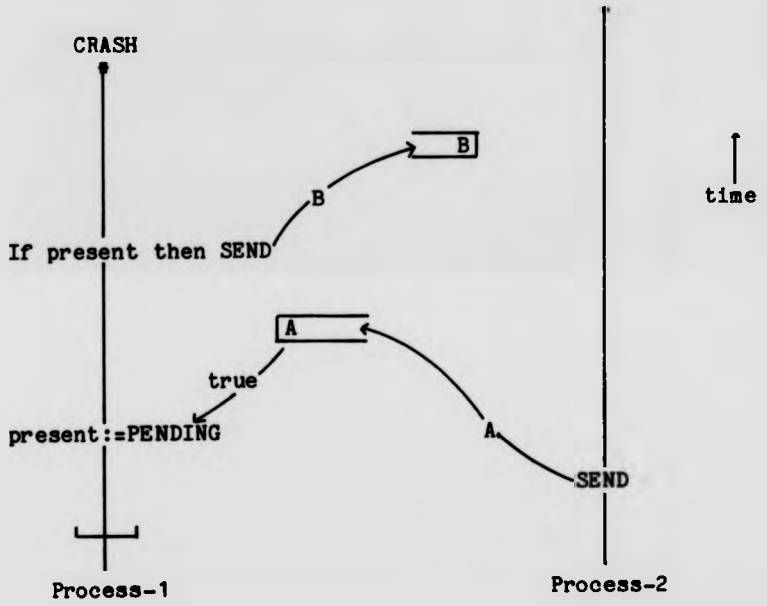


Figure 8c

PENDING is an example of a time-dependent function. A time-dependent function is a function or a procedure that when repeated by a restarted process may return a different value to the one that was returned when the command was executed during the restarted process's interrupted-task.

Restoring a restarted process to a state that is consistent with its premature messages relies on the restarted process getting the same input as its interrupted-task. Time-dependent functions input values that cannot be guaranteed to be the same when that command is re-executed after a crash.

There are a number of time-dependent functions in PROSUR's P/L. The rest of Section 8.2 identifies these commands and describes the secure point insertion strategy employed by the process survivability level in order to ensure that consistency is restored despite the existence of these time-dependent functions.

8.2.1 Time-dependent functions

There are four time-dependent functions in PROSUR's P/L. Three of these are the interprocess communication commands (SEND, RECEIVE and PENDING) and the fourth is DOIO.

a) SEND (OutputChannel, message, status, timeout)

The success or failure of a SEND depends on the length of the timeout period specified, and on how busy the network is and on how quickly the receiver is removing the messages. The network's and the receiver's activity before and after the crash will most likely be different, and so the value returned by the status parameter of a SEND that is repeated by a restarted process is likely to be different to the value that was returned when the SEND was executed during the process's interrupted-task.

b) RECEIVE (InputChannel, message, status, timeout)

The success or failure of a RECEIVE, and hence the value returned by its status parameter, depends on whether a message arrives within the timeout period or not. The message input by a repeated RECEIVE will either have been replaced by the process survivability

level, or have been re-sent by another restarted process. Even if a RECEIVE succeeded when executed prior to the crash, it is possible that when the RECEIVE is repeated the replaced or re-sent message will not arrive in time and the RECEIVE will fail. (The repeated message may never arrive as the SEND's timeout might expire.) Alternatively, if the RECEIVE first failed it may succeed when repeated.

c) PENDING (InputChannel) : Boolean

We have already shown that PENDING is a time-dependent function. The fact that PENDING actually inputs the message that it detects has no bearing on this problem.

d) DOIO (device, operation, data, status, arguments)

All DOIOs, no matter what the operation, result in some value being input either by the data or the status parameter. Devices are only connected to a single computer, and so a repeated DOIO will always fail. Assuming that the device had been working prior to the crash then this failure will be a different result to that obtained prior to the crash, and so DOIO is a time-dependent function.

Time-dependent functions may return the same value when repeated but it is not possible to rely on this, and so they must be treated as though they will always return a different result when repeated.

8.2.2 The secure point insertion strategy

Under certain circumstances it is possible for consistency to be restored despite the presence of time-dependent functions in the restarted process's interrupted-task. The following example illustrates these circumstances.

Figure 8d shows two processes at the time of a crash. Process-1 is restarted. Process-2 is unaffected and message A survives the crash. On restarting Process-1 must become consistent with its premature message A.

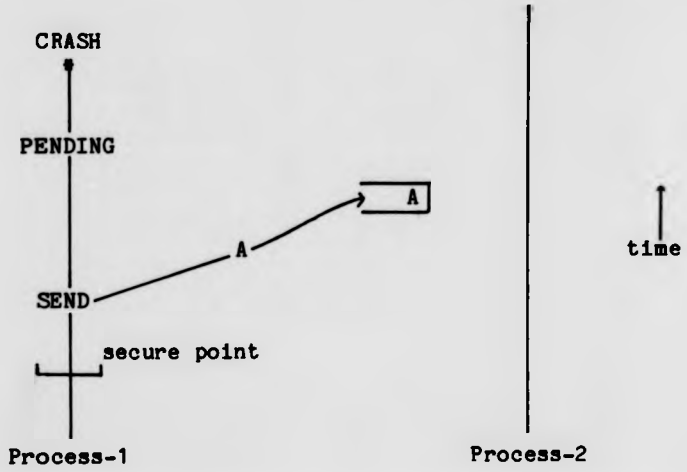


Figure 8d

Process-1's interrupted-task contained two time-dependent functions: PENDING and the SEND itself. Fortunately, the decision to perform the SEND and the composition of the message are determined before the time-dependent values of the SEND and the PENDING are available. No matter what value PENDING returns or what status the SEND returns, that SEND will be re-executed and the same message will be sent. As the SEND is a late-SEND it will be gagged. Process-1 will have recovered itself to a consistent state.

This example shows that as long as consistency has been restored prior to the re-execution of any time-dependent functions then a restarted process will recover itself to a consistent state despite the presence of time-dependent functions in its interrupted-task.

The process survivability level ensures that this is always the case by executing a secure point prior to the execution of the SEND that follows one or more time-dependent functions. As SENDs are themselves time-dependent functions, every SEND is preceded by a secure point and so a task can only contain a single SEND, and that SEND will be at the start of the task.

The secure point insertion strategy ensures that a restarted process will become consistent, but the presence of the time-dependent functions in the interrupted-task prevents the interrupted-task being completely re-executed. This does not matter, but it is necessary to redefine a restarted process's restarted-task to be those instructions of the interrupted-task that must be repeated in order for the process to become consistent. As tasks can have only one SEND in them and as that SEND is always the first instruction, a restarted-task will either consist of a single SEND or be empty.

8.2.3 Summary

Despite the replacement of missing messages it is not possible to guarantee that a restarted process will input the same values as did its interrupted-task. In order to ensure that consistency can still be restored, the process survivability level inserts a secure point before every SEND.

Performing a secure point every time a SEND is executed is a heavy overhead. A way of reducing this overhead by reducing the number of secure points that are performed is described in the next chapter.

As all tasks begin with a SEND, RECEIVEs will never be executed prior to consistency being restored and so all of our arguments presented in the later half of Section 8.1.3 appear to be unnecessary. However, in Chapter 9 we present a way of reducing the heavy secure point insertion

load by allowing a task to contain several SENDs. This optimisation also prevents a certain type of RECEIVE counting as a time-dependent function, and so these RECEIVEs may be re-executed prior to consistency being recovered and so our earlier arguments are not wasted.

8.3 Restoring consistency in a device handler

A device handler is treated by the process survivability level in exactly the same way as any other application process: a secure point is executed prior to every SEND, and when it is restarted its missing messages are replaced and its late-SENDs are gagged.

A device handler also has to be made consistent with the controlled environment's state. This requires that any input-data input by the device handler's interrupted-task be made available again so that it can be re-input, and that the DOIOs that produced the premature output-data be repeated and gagged. Unfortunately neither of these are possible.

Devices in PROSUR are only connected to single computer, and so it is impossible for a restarted device handler to control its device as its new host computer will not be connected to that device. When a restarted device handler repeats a DOIO, that DOIO always fails. Missing input-data cannot be re-input, and it is impossible to know which DOIOs to gag, and so it is impossible for a restarted device handler to become consistent with the controlled environment.

Furthermore it is not possible for a device handler to advise the rest of the control system of its device's current state as the device handler does not know what happened during its interrupted-task. The device handler only knows the state that its device was in at the time of the device handler's last secure point.

None of these problems can be removed by the process survivability level. Instead it is left to the application level to cope with them.

All of these problems could be removed by extending process survivability and PROSUR in the following ways:

1) Multi-homing the devices so that all of the host computers of a device handler's replicates are connected to its device. This ensures that a restarted device handler would still be able to control its device.

2) Enabling primary and backup replicates to exchange roles. Then if a primary replicate is unable to control its device because the computer-device link that it is using has failed, then one of its backup replicates would be activated to replace it, thereby maintaining control of the device.

3) Implementing the devices so that they can recognise repeated commands; if they receive a repeated command, the devices return the response that they returned when the command was first issued and do not execute the command. Then when a backup replicate repeats a DOIO, the DOIO would return the same data and status values as it did when it was executed prior to the crash.

A device handler's service would still be vulnerable to the failure of the device itself. The only way to remove this vulnerability is to duplicate both the device and its handler. The way in which these duplicates could be integrated into the control system was discussed earlier in Chapter 6, Section 6.2.

8.4 Time-dependent functions and secure point establishment

The presence of time-dependent functions in PROSUR's P/L dictates two restrictions on the way that secure points are executed.

Figure 8e shows two processes at the time of a crash. Prior to the crash, Process-1 sent message A to Process-2, and then it detected a message in its input channel and because of that it sent a second message (B) to Process-2. This set of actions could be the result of executing:

```
SEND (out, A, ... );
If PENDING (in) Then SEND (out, B, ... );
```

Both SENDs were preceded by secure points (SP1 and SP2).

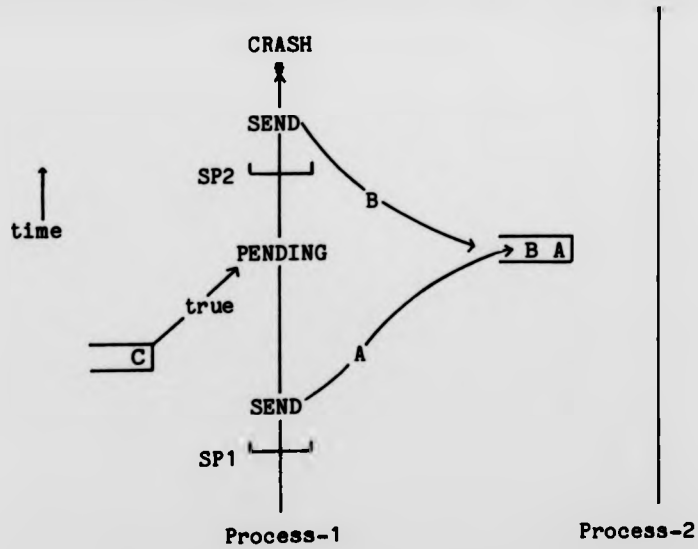


Figure 8e

After the crash Process-1 will be restarted from its last secure point (SP2) and its input channel will be empty. Process-2 is unaffected by the crash, and messages A and B will remain in its input channel. Process-1's missing message (C) is replaced and Process-1 repeats its last

SEND (which is gagged) and becomes consistent again.

However, were for some reason Process-1 to be restarted from the older of the two secure points (SP1) instead of from SP2, then it would be necessary for message C to be replaced and Process-1's two SENDs to be repeated and gagged. Unfortunately it is possible that this would not happen and Process-1 would not become consistent. For example, when Process-1 repeats the PENDING it might, for the reasons described in Section 8.2.1, return 'false' this time, and so the second SEND would not be repeated and Process-1 would not become consistent.

A description of how a secure point is established was given in Chapter 6, Section 6.2. Briefly, when a primary replicate executes a secure point, secure point data is gathered and sent by the primary replicate's host kernel to the kernels that host its backup replicates. The secure point data is then used to update the backup replicate's data segment and volatile environment.

The situation illustrated above could only arise if the last secure point had not been successfully established at the time of the crash, thus forcing the process to restart from the older, out of date, secure point. In order to prevent this from ever happening it is necessary that:

1) The primary replicate be suspended until the secure point data has been delivered to each of its backup replicate's host kernels. It is not necessary to delay the primary replicate until the secure point data has been merged with all of its backup replicates, but as a consequence of this a backup replicate must not be activated until all outstanding secure point data has been merged with its data segment and volatile environment.

2) Secure point establishment must be atomic with respect to the crashing of the primary replicate that instigated the secure point. The secure point data must either be merged with all of the backup replicates or with none of them. In the latter case, the application process will be restarted from its previous secure point, but the problem illustrated earlier will not arise as the primary replicate had not started the next task.

The two-stage protocol used to ensure that secure points are atomic is described in detail in the first half of the next chapter.

8.5 Summary

The inconsistencies that can arise due to the use of PENDING have been removed. PENDING inputs the message that it detects and stores it in the receiver's data segment from where it can be later RECEIVEd. If the record of a message's presence survives a crash so does the message.

Process recovery is supported by two actions during normal running. Firstly, every time a message is successfully sent a copy of that message is stored within the sender's data segment. Secondly, a secure point is executed prior to every SEND - this is because of the presence of the time-dependent commands (SEND, RECEIVE, PENDING and DOIO) in PROSUR's P/L.

A restarted process may have messages missing from a number of its input channels, and a single premature message may be present in one of its output channels or within the data segment of one of its receivers. The missing messages are replaced by copies of the originals. The SEND that produced the premature message during the process's interrupted-task is repeated and gagged.

Process survivability cannot restore a device handler to a state that is consistent with the results of the DOIOs performed during the device handler's interrupted-task. Furthermore, a device handler cannot report to the rest of the control system on the results of those DOIOs, or even advise the rest of the control system as to whether it performed those DOIOs or not. A way of removing both problems was presented.

The presence of time-dependent functions in PROSUR's P/L has the following implications for secure point creation:

a) The primary replicate is held up until the secure point data has been delivered to the backup replicates' host kernels.

b) Sending the secure point data to the backup replicates must be an atomic action despite the crashing of the primary replicate.

9. Implementing Process Survivability in PROSUR

9.0 Introduction

This chapter describes how the process-set organisation is implemented by the distributed kernel level, and how process recovery is implemented by the process survivability level. Process survivability's ability to provide recovery despite multiple computer crashes is also discussed.

Process survivability relies on the hardware level and the distributed kernel level exhibiting certain characteristics. The first section of this chapter recalls these characteristics.

The presence of time-dependent functions in PROSUR's P/L has resulted in a secure point being performed prior to every SEND. This appears to be a heavy overhead, and Section 9.5 describes a way in which the number of secure points can be reduced.

9.1 Hardware and distributed kernel level characteristics

In Chapter 6 we described, and rationalised, a number of characteristics that it is necessary for the hardware level and distributed kernel level to exhibit if process survivability is to be feasible. In this section we briefly recall these characteristics.

1) The network that links the computers together cannot be broken.

2) The computers are homogeneous.

3) The individual kernels are designed so that the loss of one or more kernels will not prevent the other kernels from operating normally.

4) The inter-computer communication mechanism provided by the kernels is very reliable. This mechanism is used to transmit secure point data as well as application-level messages.

5) The mapping from local pipe name to physical address of the recipient input channel is done by the sender's kernel when the message is sent.

6) Any partly sent or partly received message is discarded if the sender or receiver kernel crashes.

9.2 Process set implementation

9.2.1 Normal running

A backup replicate, like a primary replicate, has a process descriptor, an allocation of pages in which its code and data segments are stored, and has sufficient buffering for its input channels. In fact a backup replicate is managed in exactly the same way as a primary replicate, except that its status is 'unrunnable' and all of its pages are stored on the backing store.

All program addresses consist of a page number and a word-offset number pair. The primary and backup replicates' page tables define the mapping from page number to physical location and so it does not matter that the code and data segments of a primary replicate, and the code and data segments of its backup replicates, are stored in different physical locations.

A primary replicate performs a secure point by executing the kernel procedure 'ksecurepoint'. The secure point is executed by the primary replicate's host kernel which collects and sends the secure point data to the backup replicates' host kernels. Secure point data consists of:

a) a copy of the primary replicate's volatile environment;

b) the numbers of those of the primary replicate's pages which have been written to since the last secure point;

c) a copy of those pages.

Once the primary replicate has called ksecurepoint it is suspended until the secure point is completed, and so an up to date copy of its volatile environment will be contained in its process descriptor. The primary replicate program counter (part of its volatile environment) will index the instruction following the ksecurepoint and so a restarted backup replicate will be restarted from after the secure point.

When the space occupied by a page in main memory is to be reclaimed for further use, that page must first be copied onto the backing store. If that page has not been altered since it was copied into main memory, then there is no need to do this. To detect this situation page tables often include a boolean variable for each page, which is set to false when the page is copied into main memory, and is set to true when the page is written to. By extending the page table to include a similar boolean variable that is set to false for all pages when a secure point is performed and then set to true when a particular page is written to, it can be determined which pages have been written to since the last secure point.

As an alternative, tagged memory (Feustel 1973) could be used so that the secure point data would consist of those words that have been updated since the last secure point. In a similar vein Kant (1983) proposed that tagged memory could be used as an efficient way of supporting recovery blocks.

A backup replicate's code is stored in its pages at system initialisation time. There is no need for a kernel to distinguish between those pages that contain a primary replicate's code and those that contain its data, as the code pages are never written to and so will never be included in secure point data.

Once the primary replicate's host kernel has ascertained which pages have been altered, it can assemble and send the secure point data. The secure point data will be too large to be transmitted as a single block, and so it is transmitted as a sequence of smaller blocks: the volatile environment and the numbers of the altered pages are sent first, followed, one at a time, by the altered pages themselves which are first copied from where they are stored in either main memory or secondary memory.

For the reasons described in Section 8.4 establishing a secure point is atomic despite the crashing of the primary replicate's host computer: the secure point must either be completed by the senior backup replicate's host kernel or it must be aborted so that in effect it has never been executed. To achieve this the following two-stage protocol incorporating recovery is employed. (This protocol is presented as a finite state table in Figure 9a below.)

In order to simplify the following description, the primary replicate's host kernel will be called the primary-kernel, and the senior backup replicate's host kernel and the junior backup replicates' host kernels will be called the senior-kernel and the junior-kernels respectively. The senior-kernel and the junior-kernels are collectively called the backup-kernels.

state \ event	receive secure point data	receive commit	crash (senior-kernel only)
1: waiting for secure point data	store the secure point data, state:=2	ignore it	send commits to other backup kernels. FINISHED
2: waiting for a commit	discard previous secure point data	merge the stored secure point data, state:=1	discard stored secure point data. ABORT

Figure 9a

The primary-kernel sends the secure point data, in a secure point message, to each of the backup-kernels in turn. The backup-kernels are ordered by the order in which the backup replicates are designated to be activated, and the secure point message is sent to them in that order. Having done that the primary-kernel sends a commit message to each of the backup-kernels, again in the same order.

A secure point may have to be aborted, and so a backup-kernel does not use the secure point data to update the backup replicate until it receives a commit message. Each kernel has a list of the free pages on its local backing store. As a backup-kernel receives pages they are written to a free page, but the backup replicate's page table is not altered. Once the commit message is received the backup replicate's volatile environment and page table are updated and the superceded pages are freed.

If a computer crashes while a secure point message or a commit message is being sent to it, the primary-kernel aborts the transmission to that computer. The crashed computer will not be included in any further secure point.

The above is the two-stage protocol. The following is the recovery that is performed by the senior-kernel when the primary-kernel crashes. Before activating the senior backup replicate the current secure point must be completed or aborted in order to establish the same recovery point in every replicate.

First the senior-kernel discards any partially received commit message or secure point message, and then any outstanding messages are processed as normal. Once this has been done the senior-kernel completes the secure point.

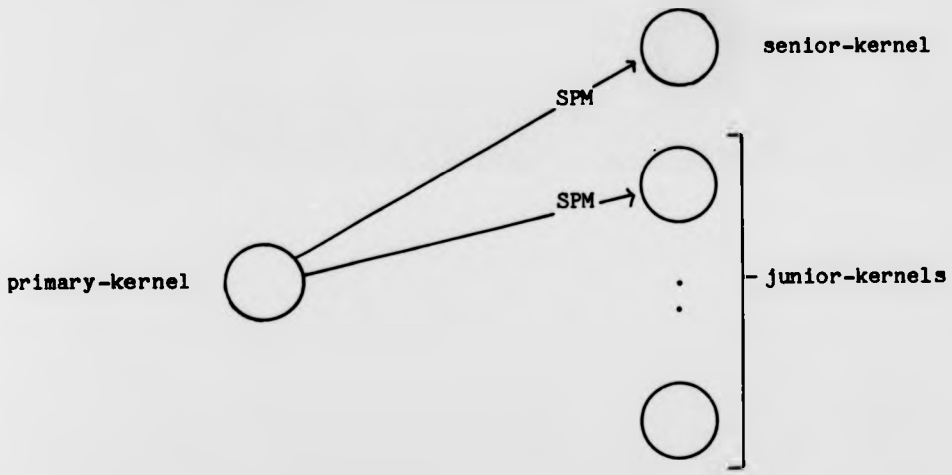


Figure 9b

Due to the secure point and the commit messages being sent to the backup-kernels in order, the senior-kernel may have received a secure point message which the junior-kernels have not, or it may have received a commit message that the others have not. Figure 9b above illustrates the former case: a secure point message (SPM) had been sent to the senior-kernel and to one other backup-kernel prior to the crash. Figure 9c below illustrates the latter case: a secure point message (SPM) has been sent to all of the backup-kernels, but the commit message (CM) has only been sent to the senior-kernel and one junior-kernel.

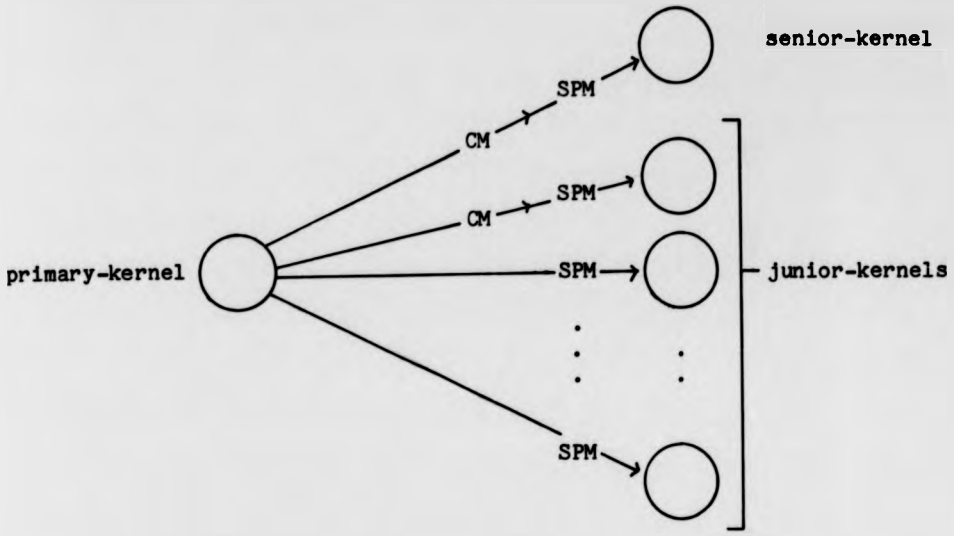


Figure 9c

If the senior-kernel has received a secure point message but not a commit, it cannot assume that the junior-kernels have all received the secure point message. The senior-kernel will discard the secure point data: the secure point has been aborted. The senior backup replicate is restarted from the previous secure point. Some of the junior-kernels may have received the secure point message, but they will discard it when they receive the next secure point message to be sent to them.

If the senior-kernel has received a commit it knows that the junior-kernels must have received the last secure point message, but it does not know whether they received a commit message, and so it sends a commit to each of the junior-kernels. This completes the secure point. The senior backup replicate is then activated. Some of the junior-kernels may have already received the commit message and they will discard the repeated one.

If the primary-kernel were to be lost between secure points, the senior-kernel would still send the commit message to all of the junior-kernels.

If the senior-kernel were to crash while finishing (completing/aborting) the secure point, then the most senior of the junior-kernels would finish it and so on, until either the secure point is finished or all of the backup replicates have been lost.

9.2.2 Error detection

Every computer has a heartbeat. A heartbeat is a small message sent at regular intervals by a computer's kernel to every other kernel. The heartbeats are monitored by the kernels. When a kernel detects that a particular computer's heartbeat has stopped it knows that that computer has crashed. The decision that a computer has crashed must be taken over several heartbeat intervals in order to prevent computers being prematurely diagnosed as having crashed just because a heartbeat was lost during transmission.

Computer crashes cannot be relied upon to be instantaneous. Often a crash will be preceded by a phase during which the computer is simply malfunctioning. If a computer malfunctions it is most likely that process survivability will be compromised and will not be able to cope with the eventual crash. To remove this threat the computers' hardware must be able to detect permanent hardware faults immediately. Having detected a fault the computer is crashed either by halting the processor or by isolating the computer from the network and its peripherals, both of which will stop the computer's heartbeat.

Ideally, the hardware should be able to recover itself from transient errors. Otherwise process survivability would have to be used to recover from a transient fault that could have been recovered from by a

simple retry mechanism.

Carter et al. (1977) have shown that the cost of producing a computer that can detect hardware faults and which has a simple retry mechanism is relatively inexpensive compared to the cost of a computer without these facilities.

9.2.3 Damage assessment

A backup replicate's predecessors consist of its process-set's current primary replicate and of those backup replicates that are designated to be activated before it. Once a backup replicate's predecessors have all crashed it is activated.

When a kernel has detected that a computer has crashed it must decide which, if any, of the backup replicates that it hosts must be activated. The information on which it makes its decision is kept in two tables.

One table records for every computer (other than the kernel's host) which of the kernel's local backup replicates has a predecessor on that computer. A second table contains a count of the number of predecessors that each of its local backup replicates has left.

When a kernel detects a computer crash it decrements the predecessor count for each of its local backup replicates that had a predecessor on that computer. If any of these counts falls to zero that backup replicate is activated.

As an example Figure 9d shows these two tables (partially completed) for a particular kernel. If computer 1 were to crash then predecessors of backup replicates 1, 2, 4, 5 and 6 would be lost. After decrementing the predecessor counts for these five backup replicates, backup replicates 1 and 4 would be activated.

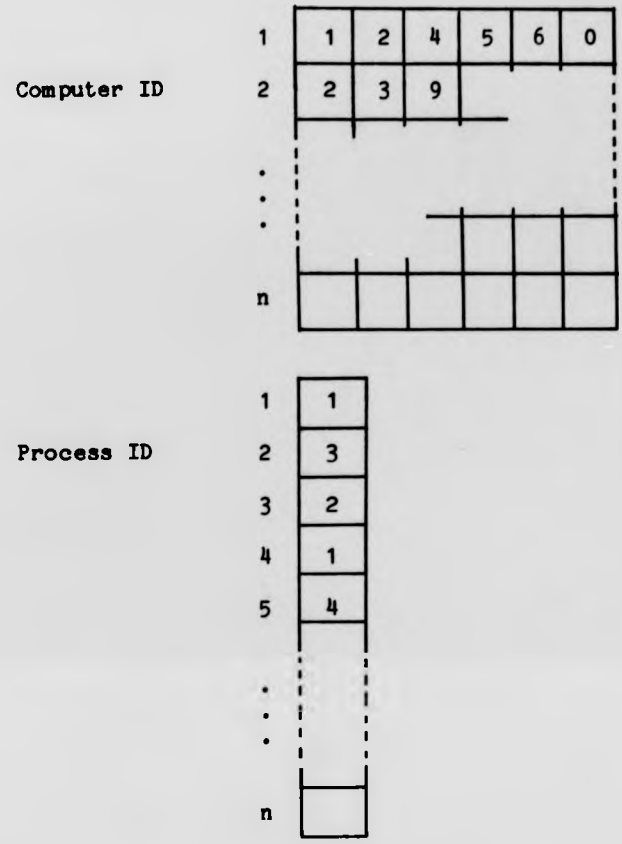


Figure 9d

9.2.4 Recovery

Having decided which of its local backup replicates are to be activated, the kernel must activate them. First however the recovery phase of the two-stage protocol described in Section 9.2.1 is performed for each of the backup replicates that are to be activated. Once that is completed the backup replicate can be activated.

A backup replicate is just a process whose state is 'unrunnable'. It can be activated by setting its status to 'runnable' and by placing its process descriptor onto the ready-queue. As the activated backup

replicate, now a primary replicate, starts to run, the resultant memory faults will ensure that its pages are loaded into the main store as necessary.

9.2.5 Fault repair and continued running

There is no fault repair. The computer that crashed cannot be re-integrated back into the control system without turning the control system off and then recreating it.

Continued running is exactly the same as normal running.

9.3 Process recovery

9.3.0 Introduction

The process survivability level defines three class types: Ichannel, Ochannel and IOdevice. (The general syntax of the class structure is given in Appendix A.) All input channels, output channels and devices defined in an application program are objects of the classes Ichannel, Ochannel and IOdevice, respectively.

When for example, an application program defines an input channel as:

```
From : Inputchannel [10] of letter;
```

this is translated into the declaration:

```
From : Ichannel
```

and into the call:

```
INIT From (10, letter);
```

The declaration declares 'From' to be an object of the class type Ichannel,

and the INIT statement creates and customises From to be an input channel of ten letters. Output channel and device declarations are translated in a similar way.

Figure 9e presents an outline of the internal structure of Ichannel, Ochannel and IOdevice. All three classes have parameters of type 'Type', which means that a single class definition can be used for all types of channel and device; for example in the Ochannel class the 'basetype' parameter defines the input channel's type. This use of a 'Type' parameter makes classes similar to generic packages in Ada (Wegner 1980).

```
Class Ichannel (size : Integer; basetype : Type);
   Export RECEIVE, PENDING;
   Procedure RECEIVE (Var message : basetype; Var status : IPCstatus;
                                    timeout : Integer);
      Begin
        .
        .
      End;
   Function PENDING : Boolean;
      Begin
        .
        .
      End;
   Begin {main body}
     .
     .
   End;

Class Ochannel (size : Integer; basetype : Type);
   Export SEND;
   Procedure SEND (message : basetype; Var status : IPCstatus;
                                    timeout : Integer);
      Begin
        .
        .
      End;
```

```
    Begin {main body}
      .
      .
    End;

Class IOdevice (device : IOdevice; basetype, argtype : Type);

    Export DOIO;

    Procedure DOIO (oper : IOoperation; Var data : basetype;
                                Var status : IOstatus; Var arg : argtype);

      Begin
        .
        .
      End;

    Begin {main body}
      .
      .
    End;
```

Figure 9e

Ichannel exports the two procedures, RECEIVE and PENDING, that can be applied to an input channel. Similarly Ochannel exports SEND and IOdevice exports DOIO. When SEND, RECEIVE, PENDING and DOIO are used in application programs, the first parameter specifies the name of the channel or device on which they are applied. As these commands are exported by classes, commands of the form:

```
RECEIVE (from, ... )
```

are translated into:

```
From.RECEIVE ( ... )
```

The internal details of these three classes are developed in the rest of Section 9.3. A program listing of the three classes (written in a Pascal-like language) is given in Appendix C.

A restarted process will have messages missing from some, if not all, of its input channels, and it may also have a single premature message within one of its output channels or within one of its receivers. To recover a restarted process the missing messages must be replaced and any late-SEND must be gagged. This recovery is performed on a per-channel basis by the input channel and the output channel at either end. There is no co-ordination between the recovery of different channels.

Recovery is performed when a channel is 'broken' by the crashing of the sender and/or the receiver process. Recovery is instigated by the input channel and performed by the output channel. Recovery is performed in this way no matter which end of the channel crashes. When a process crashes, each of its channels is recovered in this way thereby recovering the process as a whole.

The following four sub-sections describe the four stages of channel recovery.

9.3.1 Normal running

Ichannel is based around a kernel-supplied input pipe called DataIn, and Ochannel is based around a kernel-supplied output pipe called DataOut. An interprocess channel is created by linking DataIn and DataOut together at system initialisation time, and the interface to this pipe is provided by the input channel and output channel class object at either end. The rest of Ichannel's and Ochannel's attributes are concerned with the provision of process recovery after a crash.

A class object is part of its host process. When a process executes a secure point the private data of all of its objects are also backed up. Thus the information on which recovery is based survives the crash, and this information will be consistent with the rest of the restarted process's state as it was backed up at the same time.

As explained in Chapter 8, Section 8.1.2, the PENDING-related inconsistencies are prevented by implementing both PENDING and RECEIVE in a particular way and by extending the input channel to include a single-message buffer. This buffer is called 'Pstore', and the boolean 'Pflag' is used to indicate whether it is full or not. Both Pstore and Pflag are declared within Ichannel.

When PENDING is called it checks whether there is a message in Pstore; if there is, PENDING returns true as there is a message in the input channel; if on the other hand Pstore is empty, PENDING attempts to input a message from DataIn using the procedure 'Read' with a zero timeout. Because of the zero timeout a message will only be input if it is already present in the pipe (Chapter 6, Section 6.3 discusses zero timeouts). If the Read succeeds, then the message is placed in Pstore, Pflag is set to true and PENDING returns true, but if Read fails, then Pflag is set to false and PENDING returns false. When RECEIVE is called it will return Pstore's contents if Pstore is full (and it will set Pflag to false), but if Pstore is empty, then a message is input normally using Read.

In Chapter 8, Section 8.1.2 we mentioned that when calculating which messages survived a process's crash it is necessary to take into account the fact that PENDING actually inputs the message that it detects. RECEIVE and PENDING both use the procedure Read to input messages, and so when a process crashes it loses those messages that are buffered within its input pipes (the DataIn in each of its input channels) and the messages that it has Read in since its last secure point. Those messages that were Read in prior to the process's last secure point will survive.

All of the code at the input channel end for supporting channel recovery during normal running is contained in Read. The code in RECEIVE and PENDING is solely concerned with preventing the PENDING-related inconsistencies from arising.

Returning to channel recovery, there are three activities performed during normal running in order to ensure recovery. These are:

1) secure point insertion;

2) message storage;

3) message tagging.

Secure pointing is provided by a class type called SPoint. Every application process has a single object of this class called SP. A listing of SPoint is included in Appendix C.

SP exports a procedure called 'Execute'. When Execute is called it uses the kernel procedure ksecurepoint to perform the secure point. The reason for embedding the call of ksecurepoint inside a class is that SP also performs important housekeeping operations, details of which are given later.

In order to ensure correct recovery, a secure point must be executed prior to every SEND, and so the Ochannel procedure SEND calls SP.Execute before attempting to actually send the message.

When a channel is broken messages may become missing. In order to replace these messages with copies of the originals it is necessary to store copies of the originals. This is done by the output channels themselves.

SEND uses ksend to output the message. If ksend is successful (status = succeeded) a copy of that message is placed onto the cyclic queue 'RcvyStore' by calling the procedure 'save'.

It is necessary to limit RcvyStore's size. To do this the maximum number of messages that can possibly be replaced at recovery time must be limited. This in turn means that the number of messages that can be lost must be limited. RcvyStore's size can then be set to that limit.

The messages that are lost from a channel when its receiver crashes are those still in the input pipe and those that had been Read from it since the receiver's last secure point. The former is limited by the size of the input pipe which is defined by the class parameter 'size', and the latter is limited by the input channel executing a secure point after a certain number of successful Reads.

Every time a message is successfully input Read increments the Ichannel variable 'Rcount'. When Rcount becomes equal to the maximum 'mRcount' a secure point is performed. The maximum number, mRcount, is arbitrarily set to the defined size of the channel which is the class parameter 'size'.

Thus the maximum number of messages that can be lost from a particular channel is 2*size, and so RcvyStore's size is 2*size. There is no explicit flow control for placing messages into RcvyStore. If the message has been sent, then the space existed in the channel and so the copy can be stored without overwriting a message that might still be needed for recovery.

When a receiver crashes it is restarted with an empty input pipe and then the missing messages are replaced. The maximum number of messages that might have to be replaced is twice the size of the channel, and these would not all fit into the input pipe DataIn. So a further requirement is to ensure that all of the lost messages will fit into the input pipe.

To do this, the size of the input pipe DataIn is increased from 'size' to 'extsize' which is equal to the maximum number of messages that can be lost, i.e. 2*size. However, it is then necessary to ensure that this does not mean that the number of messages that can be lost is also increased. This is why, when kread (see Chapter 6, Section 6.3) returns a message from an input pipe, it does not release the space that the message occupied. Hence when a message is Read (using kread) another message cannot be sent to take its place, and so the maximum number of messages that can be lost remains 2*size.

Once a secure point has been performed the Read messages cannot be lost and so the space they occupy can be freed. This is done by calling the kernel primitive kfree which releases the space occupied by messages that have been kread from that input pipe since the last kfree.

Any secure point, executed by any of a process's input channels and output channels, will prevent the loss of all of the messages Read from all of that process's input pipes since the last secure point. This fact is used to reduce the secure pointing frequency.

Every time a secure point is performed the secure pointing class SPoint increments the value of its variable 'tstamp'. The current value of tstamp can be read by using the function 'Timestamp' which is exported by SPoint. This value is used by an input channel to identify individual secure points so that it can detect whether a secure point has been executed since it last Read a message. Input channels remember the last secure point's identity (SP.Timestamp) in 'timestamp'.

When Read is called the value of timestamp is compared with that of SP.Timestamp. If a secure point has been performed then kfree is called, timestamp is updated and the count of the number of Reads performed is reset to 0.

In order to perform channel recovery it is necessary to calculate which messages are missing and which, if any, is premature. To facilitate this every message sent down a channel is labelled with a sequence number.

Each output channel maintains a count ('Sseqno') of the number of times it has been called upon to send a message. Every time SEND is called, Sseqno is incremented and its new value is used to tag the message sent. Every time a message is Read by an input channel the sequence number is stripped off and stored in 'Rseqno'. The use to which Rseqno and Sseqno are put is described in Section 9.3.3.

A channel's type is defined by the parameter 'basetype' which is passed to both input and output channels. To accommodate the sequence number the output pipe's type is extended from 'basetype' to 'extbasetype'.

9.3.2 Error detection

Every input channel class and every output channel class has an exception handler called 'Recovery' which is associated with an event. When such an event occurs the exception is said to have been raised and the exception handler associated with it is executed. To execute an exception handler the host process's execution is interrupted and control is passed to the handler. When the handler has finished, control returns to the point at which the host process was interrupted. This is different from accepted practice (for example Cristian 1980, Goodenough 1975 and Harland 1981) where the interrupted code is aborted, and is closer to the software interrupt used in interprocess communication in DSN (Rashid 1980). By using exceptions in this way recovery can be performed when necessary without disturbing the work being done by the processes.

Damage assessment and recovery is performed by the output channel. To do this the output channel needs to know:

a) the sequence number of the last message to be sent successfully down that channel;

b) the sequence number of the last message to be delivered successfully to the input channel and to survive the crash;

and also it needs to know when to perform the recovery.

The output channel already knows the first of these values as it is stored in Sseqno. The second value, as well as the prompt, is supplied by the input channel at the other end of the channel.

When a channel breaks - due to the sender and/or the receiver being restarted - the input channel's exception handler is executed. Detecting the channel break and raising the exception is performed by the receiver's current host kernel. Details of how the kernel does this are given in Appendix B.

Ichannel's exception handler determines the sequence number of the last message to have been successfully sent down the channel and to have survived the possible restarting of the receiver. If the input pipe is not empty then this message is the most recently delivered message in the input pipe, and if the input pipe is empty then the message was the last one to be Read. The sequence number in the latter case is contained in Rseqno and in the former case the sequence number can be obtained using kpeeklast (defined in Chapter 6, Section 6.3). Kpeeklast is called with a zero timeout so that it will only succeed if a message is already present in the input pipe. If kpeeklast succeeds then the sequence number of the message returned is the required sequence number; if not then Rseqno is the required number.

The input channel and the output channel are linked by a second pipe which goes from the input channel where it is called 'RcvyOut' to the output channel where it is called 'RcvyIn'. The sequence number is sent by the input channel to the output channel down this pipe. However before the message can be sent the pipe must be reconnected.

A process's descriptor contains a table that defines for each of its output pipes the address of the recipient input pipe (see Chapter 6, Section 6.3). This address contains the number of the input pipe's host computer. After a crash this computer number will be incorrect: if the sender crashed then on restarting the location will be set to zero, and if the receiver crashed then the receiver will be on a different computer to that specified in the table. Hence after a crash output pipes must be reconnected so that the table contains the current computer locations of the recipient input pipes.

The output pipes that are used to carry recovery messages (RcvyOut) are reconnected by the input channel to which they belong and the output pipes that are used to carry user messages (DataOut) are reconnected by the output channel to which they belong. There is no need to reconnect the input pipes as they will accept messages from any source.

When an input channel's exception handler 'Recovery' is called it is passed two parameters by the kernel. One ('rloc') is the input channel's current computer address, and the other ('sloc') is the current computer address of the output channel. Sloc is used by Recovery in a kconnect call in order to reconnect RcvyOut. Rloc is sent, along with the sequence number, down the reconnected pipe to the output channel where it can be used to reconnect DataOut.

When the recovery message arrives in RcvyIn the input channel's exception is raised and its exception handler is executed. That is the end of the error detection phase.

When an exception is raised it will be handled as soon as the current instruction being performed by its host process has been completed. If the current instruction is a kernel procedure that involves a wait, for example ksend, kreceive, kdoio and kread, then the exception is handled in parallel to the wait, thereby preventing the exception handling being held up. The other kernel procedures such as kconnect and ksecurepoint are completed before the exception is handled: an important feature as ksecurepoint requires the process to be inactive.

In effect a process and its exception handlers are executed concurrently. This concurrency is virtual rather than actual as the process is suspended while its exceptions are performed. This virtual concurrency results in the need for critical regions (Brinch Hansen 1973) to be created in order to prevent the concurrent accessing of shared data. For example, in an input channel both Read and Recovery access Rseqno, and in an output channel both SEND and Recovery access Sseqno, ngagged and RcvyStore. To prevent concurrent access of this data these procedures and the exception handlers are executed with exception handling turned off (achieved using the 'With ExceptionsOff Do' construct). While a process is within one of its input channel's or output channel's critical regions, exceptions raised within that class object will not be handled until the process has left the critical region.

As an object's exception handler cannot be executed while control is within its critical region (within its handler or its procedure) it must be ensured that the process will eventually leave the critical region. To facilitate this any outstanding or future kernel call that involves a wait such as ksend will be aborted with a status of failed. However, an

exception raised within one object does not result in the aborting of kernel procedures executed by another object. For example, an exception raised in one input channel will not cause a kread issued by another input channel to be aborted.

If the kread at the heart of the RECEIVE or the ksend at the heart of the SEND is aborted in this way then the result is passed back to the process that called the RECEIVE or the SEND. Recovery and computer crashes are not totally invisible.

9.3.3 Damage assessment

Damage assessment is performed by the input channel's exception handler Recovery. Recovery uses the sequence number received from the input channel and the sequence number contained in Sseqno to decide whether messages need to be replaced, or whether SENDs need to be gagged, or whether no recovery is needed at all.

If the sequence number of the last message to arrive in the input channel is less than the sequence number of the last message to be sent then messages may be missing and so must be replaced. If the sequence number of the last message to arrive is greater (by one at the most) than the sequence number of the last message sent then the next SEND performed on this channel must be gagged. If the two numbers are the same then, with respect to this channel, the sender and receiver are consistent and no recovery is needed.

9.3.4 Recovery

All of the recovery is instigated by the output channel's exception handler Recovery. First it reconnects DataOut using kconnect specifying the computer number sent to it by the input channel. Then if message replacement is needed it calls the procedure Replace, or if gagging is needed it sets 'ngagged' to the number of SENDs that must be gagged (Sseqno

- rm.seqno, which will be either 1 or 0).

Replace takes copies of the missing messages from RcvyStore and re-sends them in the order in which they were originally sent. The copies are not removed from the queue as they may still be needed in case of future crashes.

Sseqno, the value used to tag messages, is incremented every time SEND is called. It is not a count of the number of messages sent. Thus sseq-rseq is the maximum, not the actual, number of messages that could be missing. So before re-sending a message, Replace checks that the message's sequence number is within the range of those missing.

A SEND can fail either because its timeout expired or because an exception was raised within its input channel. As explained in Section 6.3, although a SEND has failed it is possible that the message has been delivered. Some of the messages that are missing after a crash may be the product of failed SENDs, but they will not be replaced as there will not be a copy of those messages in RcvyStore. This is an advantage as it removes these inconsistencies.

Recovery sets ngagged. The actual gagging is performed by SEND itself. When SEND is called it checks the value of ngagged to see if there are any SENDs still to be gagged. If there are it simply sets its status parameter to succeeded, decrements ngagged and returns; it does not ksend the message. The SEND has been gagged.

A SEND that returned a status of 'failed', but actually transmitted the message, may be gagged if that SEND was performed during the restarted process's interrupted-task and if the resultant message survived the crash. As a SEND is a time-dependent function its outcome will not affect correct recovery, and so it does not matter that the status returned by the SEND when it is re-executed is different to the status returned when it was

first executed.

Gagging is only performed by restarted processes. If a process has not been restarted, recovery only consists of message replacement.

When a backup replicate is activated all of its input channel exceptions are raised and will be handled before the restarted process starts to execute its normal code. Its output channel exceptions however will not be raised until the receivers send their recovery messages. Because of this delay it is possible that a SEND may be performed on a channel that has not yet been recovered. Such a SEND will fail because DataOut will not have been reconnected yet. Once that output channel's exception has been handled it may be necessary for the SEND to be gagged, but it would then be too late. To prevent this it is necessary to hold SENDs up until the output channels on which they operate are recovered. This is done using insecure variables.

An insecure variable is a boolean variable which has a defined default value to which it will revert after a crash. Every output channel has an insecure variable called WaitRcvy. WaitRcvy has a default value of true. When the class is created using INIT, WaitRcvy is set to false. After a crash all of the process's output channels' WaitRcvys revert to true. When an output channel's exception handler has finished it sets WaitRcvy to false. SENDs are delayed by the 'Waituntil' statement until WaitRcvy is false; thus a SEND performed before its output channel has been recovered is delayed until that channel has been recovered.

An insecure variable is stored as part of the process's process descriptor, but would not be backed up by a secure point. A backup replicate's insecure variables would be set to the default value.

9.3.5 Continued running

Continued running is the period from when a backup replicate is activated, up until all of its channels have been recovered, at which point the restarted process enters its normal running phase again.

During the continued running phase a restarted process will execute its code while at the same time being interrupted by the performance of its exception handlers. This section describes the effects that performing recovery has on such a process. It also describes the effect that supporting recovery has on an unaffected process that is connected to one or more restarted processes. One of the aims of process survivability is that it should be invisible to the processes, and so this section is, in part, a review of how successfully this aim has been met.

An unaffected process continues to execute its code uninterrupted except for the execution of exception handlers. If the unaffected process performs a SEND, or if a SEND is outstanding on an unrecovered channel, then that SEND will fail because the pipe DataOut has been broken and has not yet been reconnected. The SEND is not delayed by the WaitUntil statement as the process has not been restarted.

If an unaffected process performs a RECEIVE on an unrecovered channel, then the RECEIVE will be aborted with a status of failed if that input channel's exception is raised even though messages are present in the input channel. If the RECEIVE is not interrupted then it will act as normal.

When a backup replicate is activated all of its input channel exceptions will be raised, and they will be handled before the restarted process starts to execute its code. Any RECEIVE will be issued on a recovered channel and so will not be aborted due to an exception being raised. However, the RECEIVE's timeout period may expire before the

missing messages are replaced, and so a RECEIVE may fail when it would have succeeded had it not been for the crash.

SENDs issued by the restarted process on an unrecovered output channel will be delayed until the channel has been recovered. As the Waituntil is outside the SEND's critical region, the SEND will not be aborted when its output channel's exception is eventually raised.

While the restarted process is in the continued running phase it also performs the operations performed during normal running in support of process survivability.

Process survivability is not totally invisible. In unaffected processes, SENDs and RECEIVEs fail that would otherwise have succeeded, and in restarted processes RECEIVEs that would otherwise have succeeded may fail and SENDs may be delayed past their timeout period. When a RECEIVE or SEND fails due to an exception being raised the process will treat it as if the timeout had expired as the two causes are indistinguishable. The major problem is extending the time taken to perform a SEND beyond its timeout.

Computers will not crash very often, and so the interruptions caused by process survivability are comparatively minor, especially when compared with the advantages of process survivability.

9.4 Process survivability and multiple crashes

One of the stated design aims of process survivability is that it should be able to restore consistency despite multiple simultaneous, or nearly simultaneous, computer crashes. The computers are physically dispersed so as to reduce the chances of simultaneous crashes, but the possibility cannot be ruled out and so process survivability must be able to cope with this eventuality.

Process-sets are based on multiple backup replicates so that as long as the number of computers that crash is less than the level of redundancy, processes are not lost. Also the two-stage protocol used to send secure point data to the backup replicates is designed to establish a common secure point in all the replicates despite multiple crashes.

This section shows that the process survivability level is able to restore the processes to consistent states despite multiple crashes.

Simultaneous crashes are defined on a per-channel basis to be the crash of two or more computers such that the sender or the receiver process crashes while the exceptions raised by a previous crash are still raised or being handled. If the crash occurs after the class's exception handler has finished then the crashes are separate crashes.

The implementation described in the previous section can handle separate crashes. The fact that a separate crash has occurred earlier does not affect the handling of a later crash. Message replacement does not remove the message copies from RcvyStore and so they can be used again, and a gagged SEND stores a copy of the gagged message in RcvyStore so that it can be replaced if necessary. The rest of this section concentrates on simultaneous crashes.

A kernel detects that a computer has crashed when that computer's heartbeat fails. Different computers' heartbeats are staggered, as they cannot use the network at the same time, and so even if two computers crash at exactly the same instant, a kernel will detect one crash before the other. Hence a sender and a receiver never crash together.

We look first at the effect of simultaneous crashes on input channel recovery actions. Every time the kernel detects that the receiver's primary replicate has crashed or that the sender's primary replicate has crashed then the exception is raised in the input channel.

If the receiver has crashed then the exceptions will be raised in different generations of the input channel. Where an exception has been raised a number of times the different instances are handled in the order in which they were created.

When the sender or the receiver crashes the input channel may:

1) already have its exception raised due to a previous crash;

2) be handling its exception;

3) already have handled a previous instance of the exception but recovery has still not been performed by the output channel to which the recovery message was sent.

The following describes the effect that the receiver or the sender crashing has on the actions of an input channel in each of the above situations. A combination of these three situations may apply; for example, an input channel may be handling one instance of its exception with another instance of that exception outstanding. The effect that a crash has in such a situation will be the union of the effects arising from the individual situations as described below.

1) <u>The exception is already raised</u>

a) receiver crashes

All of the outstanding instances of the exception are lost in the crash and so will not be handled.

b) sender crashes

The outstanding instances of the exception will be handled in turn. These outstanding instances are the result of the sender crashing several times before and so the computer location of the sender specified in each exception instance will be out of date, and so the sending of the recovery message will fail.

2) A previous instance of the exception is being handled

a) receiver crashes

The exception handling is terminated by the crash, but the recovery message may have been successfully sent prior to the crash.

b) sender crashes

The recovery message may have been successfully delivered before the sender crashed. If not then sending the recovery message will fail.

3) After a previous instance of the exception has been handled

No matter whether the sender or the receiver crashed the crash has no effect on the results of the previous exception - the recovery message has been sent.

On restarting (if the receiver was hit) or on handling all outstanding exceptions (if the sender was hit) the input channel class will handle the exception raised because of the last crash, and the recovery message will be sent to the output channel.

In the above there are examples of instances of the exception being lost before a recovery message could be delivered to the output channel class, and so multiple crashes do not always result in multiple recovery messages being sent to the output channel. An output channel is not always made aware of multiple crashes.

Alternatively, there are situations where simultaneous crashes do result in multiple recovery messages being sent to the output channel. Each recovery message contains data that was up to date when the exception was handled, and these messages will arrive in the order in which they were sent. The following describes the effect that a crash and the arrival of the resultant recovery message have on an output channel that is performing recovery.

In the following we describe the effect that a crash has on recovery, and argue that interrupting recovery in this way does not affect the output channel's ability to perform recovery correctly. We first describe the effect of the receiver crashing, and then the effect of the sender crashing.

1) Receiver crashes

If the recovery being performed is gagging then it will not be affected by the loss of the receiver as it does not involve DataOut. If the output channel is performing message replacement at the time of the receiver's crash (or if the receiver crashed before it started) then those messages sent by the procedure 'replace' before the crash will have been delivered, but any ksends performed after the crash will fail.

The recovery message that was generated due to the receiver's crash will arrive. As the receiver crashed the only recovery that could be needed is message replacement. Message replacement prior to the crash will not have affected the contents of RcvyStore, and if a SEND had been gagged then a copy of that message will have been stored and so can be replaced.

2) Sender crashes

Any recovery that was being performed is ended by the crash and any outstanding exceptions will be lost. When the sender is restarted its recovery data will have been reset as well and so will not be left in an inconsistent state by the crash. The recovery message sent as a result of the sender crashing will arrive, and the output channel will handle the exception as normal.

In the above we have shown by case analysis that process survivability can restore the processes to consistent states despite multiple computer crashes, even if those crashes interrupt the recovery

performed by the input channel's and output channel's exception handlers.

9.5 Reducing the number of secure points needed

The presence of time-dependent functions in PROSUR's P/L means that every SEND must be preceded by a secure point. Such a secure point frequency appears to be heavy (although it is the same as the checkpoint frequency in Tandem), and in this section we present a way of reducing it.

The major problem is that SEND itself is a time-dependent function and so there can never be more than one SEND per task. The reason that SEND is a time-dependent function is that it includes a timeout. If the timeout were removed then SEND would no longer be a time-dependent function and there could be several SENDs in a task, and hence there would be less secure points.

A particular job will often be performed by a group of processes working together on the same computer. This association of processes has been recognised and has been used explicitly to structure the organisation of the application level in a number of distributed systems (Liskov 1979, Kramer et al. 1982).

We propose that processes can be grouped together to form a module. All processes in a module are on the same computer. Processes within a module are connected by intra-module channels, and processes in different modules are connected by inter-module channels. Intra-module channels are also intra-computer, but inter-module channels can be either inter-computer or intra-computer.

The backup replicates of processes in the same module are located on the same computers and in the same order. This ensures that after a crash a module's processes are still on the same computer, and so intra-module channels remain intra-computer despite crashes. Inter-module

channels may alter between being inter-computer and intra-computer.

Transferring a message between computers may take a long time because the network may be busy and because delivering the message correctly could (in theory) take an infinite amount of time. To prevent this delay affecting a sender process's response time the SEND command has a timeout. If a message is being sent intra-computer then there is no such delay, and so there is no need for the timeout. By omitting timeouts and status parameters on intra-computer SENDs, those SENDs are no longer time-dependent SENDs.

We propose that all SENDs and RECEIVEs on intra-module channels do not have timeout and status parameters. Hence these SENDs and RECEIVEs are not time-dependent functions.

An intra-module SEND that succeeded during a process's interrupted-task will succeed when repeated, as there is no timeout to prevent it. An intra-module RECEIVE that was executed during an interrupted-task will succeed and input the same message, as (having no timeout) it will wait for the missing message to be replaced or for the repeated message to be sent by the intra-module SEND which will succeed as it has no timeout either.

Furthermore, intra-module SENDs and RECEIVEs cannot fail due to an exception being raised. As intra-module channels are also intra-computer both sender and receiver will be restarted. The receiver's input channel exceptions will be handled before it starts to re-execute its interrupted-task, and so intra-module RECEIVEs cannot be interrupted by an input channel exception being raised. The sender is restarted as well and any re-executed SEND will be delayed on its 'Waituntil' statement until after that output channel's exception has been handled and so the SEND will not be interrupted by that exception.

Inter-module SENDs, inter-module RECEIVEs, DOIOs and PENDINGs are still time-dependent functions. Intra-module SENDs and RECEIVEs are not time-dependent functions. All SENDs, whether intra-module or inter-module, must be separated from preceding time-dependent functions by a secure point. A task can now contain a number of intra-module SENDs, but still only only one inter-module SEND. As it is likely that the amount of intra-module communication will be higher than the amount of inter-module communication, being able to have several intra-module SENDs in a task should result in a good reduction in the number of secure points.

In the simulation presented in the next chapter we compare the performance obtained using this module-based secure point insertion strategy with the performance obtained using the 'worst-case' policy where every SEND causes a secure point.

The loss of timeouts from intra-module SENDs and RECEIVEs may be a problem as they are also a way of preventing a process from being held up by a sluggish sender or receiver. However, as a module is a single entity and would be coded as such, this problem may not arise. The simulation study should give an indication as to whether this sacrifice is worth the reduction in the number of secure points that are performed.

9.6 Summary

In this chapter we have shown how the process-set is implemented by the distributed kernel level, and how process recovery is performed by the input channels and the output channels. Furthermore it has been argued that process survivability can cope with multiple computer crashes.

Supporting process survivability requires a secure point to be executed prior to every SEND. We have presented an alternative to this, based on the adoption of modules, which should considerably reduce the number of secure points performed.

The process survivability level has been presented in earlier chapters as providing both the process-set and process recovery. However, the former task is performed by the distributed kernel level. The process survivability level consists of the input channels and the output channels, and only implements process recovery, but it remains convenient to continue presenting process-sets as being one of the services that the process survivability level provides.

10. A Simulation of Process Survivability

10.0 Introduction

Process survivability is practicable, but is it practical? In the previous chapters we have argued that process survivability will work, but we do not know whether the overheads incurred will prohibit its useful adoption. A discrete event simulation of PROSUR has been implemented in order to investigate the effects of these overheads.

The simplification and generalisations which of necessity have been introduced into the simulation prevent the results of the simulation being used to determine finally and irrevocably the practicality of process survivability. Instead it is intended that the results will provide us with an insight into the probable effects that the addition of process survivability would have on a distributed computer control system's performance.

10.1 The aims of the simulation

In this simulation the useful work performed by an application process is measured by the number of seconds for which that process was actually being executed (CPU seconds) in a given period of real time. A distributed computer control system's performance is then measured by the average of the amount of useful work performed by each of its application processes.

As response time is an important factor in real-time systems it would arguably have been more useful to have used the average response time of the application processes as a measure of the system's performance. However, in order to measure this an actual application would have had to been simulated, and unfortunately despite an extensive literature survey and letter writing campaign we were unable to find any suitable details for

an existing system, and it was decided that it would be impractical in the time available to invent a realistic application. In the summary to this chapter, we attempt to draw conclusions about how the response time is affected.

A distributed computer control system will spend almost all of its entire working life in the 'normal running' state. The effect that process survivability has on a system's performance during normal running would to a great extent determine whether it could be usefully adopted or not. For certain applications a further important (if not critical) consideration would be the length of time that it takes the system to recover from a crash. However, important as the latter is, we have limited the simulation to the investigation of the effect that process survivability has on normal running performance in order to keep the simulation within manageable proportions.

To achieve our aim the following aspects of process survivability have been investigated:

a) In the previous chapter we described two secure point insertion strategies. One was worst case, requiring a secure point before every SEND. The other was an optimisation based on the use of modules within PROSUR's P/L which should lead to a reduction in the number of secure points and possibly to a reduction in the overheads as well. Both insertion schemes are modelled and the comparison of the results will enable us to gauge the advantages that might result from adding modules to PROSUR's P/L and from the adoption of the optimised insertion strategy.

b) The level of redundancy will increase the amount of work involved in performing a secure point thus increasing the overheads. The way in which performance is affected by different levels of redundancy is investigated.

c) In both secure point insertion strategies it is the SENDs that determine the position of the secure points. An increase in the frequency with which the processes perform SENDs will increase the frequency of secure points and hence the overheads. The effects of varying the SEND frequency are investigated.

d) An accepted (if expensive) way of increasing a distributed computer control system's performance is to increase the number of computers. We investigate whether such an increase is maintained after the addition of process survivability, and whether increasing the number of computers is a way of maintaining performance after the addition of process survivability.

The model of PROSUR that was simulated is described in the following two sections. The experiments performed and the results obtained are described in the last four sections of this chapter.

10.2 The simulated distributed computer control system

10.2.0 Introduction

To simulate PROSUR in full detail would be a prohibitively large task. Instead a simplified version of PROSUR has been modelled. This section describes this simplified version.

Process survivability overheads during normal running involve a large amount of network activity in order to send secure point data to the backup replicates' host kernels. The network is a shared resource and so an increase in its use will affect the whole of the control system. The simplification of PROSUR is achieved by concentrating on the network related parts, so that the network activity is modelled realistically, and by simplifying the rest.

10.2.1 Hardware

The hardware configuration is shown in Figure 10a. The computers are linked together by a Cambridge Ring (Wilkes and Wheeler 1979). The Cambridge Ring was chosen because of familiarity with its operation. Each computer is interfaced to the ring by a locally designed access logic unit (Bennett and Singleton 1982). The access logic unit (ALU) is a microprocessor based unit that implements the basic block protocol (Johnson 1980) thereby presenting the computer with a high level interface to the network.

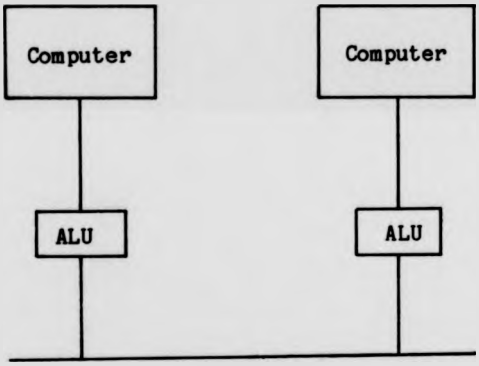


Figure 10a

An ALU receives messages from its computer that are to be sent over the network. These are placed in output queues, one for each possible destination computer. The ALU serves its output queues in round-robin order sending a message from each in turn. The message to be transmitted is placed into a basic block (with a 3 word overhead) and then sent to the destination computer's ALU. The speed with which a basic block can be transmitted is a function of the ring's raw data rate and the number of network nodes.

The Cambridge Ring's raw data rate is 10 megabits/sec. Packet overheads and sharing result in each ALU having a guaranteed point-to-point share at the physical level equal to 4/(n+1) megabits per second, where n is the number of ALUs (Bennett and Singleton 1982). Throughout the simulation we use a 16 bit word, and so this data rate is equal to 1/(4*(n+1)) mega words per second. However, the maximum data rate that can be supported by an ALU is 50 Kwords per second, and so the guaranteed minimum data transfer rate between two ALUs at the physical level is:

1/(4*(n+1)) Mwords per second, for n > 4

and

50 Kwords per second, for n <= 4.

This data rate is a worst case minimum, as it is based on the assumption that every ALU is using the network as much as possible. In 'real life' some of the ALUs would not be using their full share, and this unused network capacity would be available to the other ALUs. However, in the interests of simplicity we have used this worst case figure as the maximum rate at which data can be transferred between ALUs despite the inactivity of other ALUs.

An ALU can only receive one basic block at a time as on receiving a basic block's header it sets the source select register of the Cambridge Ring station to the sender's address (Johnson 1980). If an ALU cannot deliver a basic block because the receiver is already receiving a basic block from elsewhere, it waits for 16 cycles (where a cycle is the time it takes to send one word over the network) then tries again. If it still cannot deliver the block, it gives up trying to send that block, leaves the message on its output queue, and continues to handle its output queues normally. (16 was chosen as any lesser number resulted in the simulation overrunning the available runtime.)

10.2.2 The application level

We first describe the model used to represent all processes, and then describe how the processes are configured to form the application level.

Processes are grouped into modules. Although modules are not part of PROSUR they have been added in order to investigate the module-based secure point insertion strategy.

Each process is modelled as a RECEIVE-work-SEND cycle. The work period, measured in CPU seconds, is generated by a Normal distribution with mean m, standard deviation 1 and a truncated range of 0 to 2*m. All processes have the same mean.

A RECEIVE is satisfied by a message from any of the calling process's input channels. This corresponds to the best case where a process never has to wait for a message from one channel when messages are outstanding on other channels. (Were the channels to be specified it would be possible for the simulated control system to deadlock.)

Before a message can be sent the output channel must be chosen. A module consists of processes that work together to provide a service. Hence it is envisaged that most message passing will be intra-module, and it was decided that 80% of all message passing is intra-module and only 20% inter-module. Having determined whether an inter-module or intra-module output channel is needed, the specific channel to be used is chosen by round-robin. The sender is suspended until the message is safely delivered into the receiver's input channel.

Messages do not have any contents but they do have a size. Based on the survey presented by Prince and Sloman (1981), it was decided that there would be two types of message: command (4 words long) and data (256 words long), and the ratio of command messages to data messages would be

Flow control on channels is only partly modelled. A receiver will be hung up if there are no outstanding messages in any of its input channels and will be activated upon the arrival of the first message in any input channel. There is no flow control on the SENDS; it is assumed that input channels never fill up. Timeouts on RECEIVES and SENDS are not modelled.

For our purposes a distributed computer control system configuration consists of C computers, each containing M modules of P processes each. Every process has R inter-module output channels and L intra-module output channels, where C, M, P, R and L are constants for a particular configuration.

Within a module the processes are ordered cyclically and a process with L intra-module output channels will be connected to the next L processes in the circle excluding itself. For example, Figure 10b shows the internal structure of a module where P equals five, and L equals two: the five processes (circles) are each connected by two intra-module output channels (directed lines) to the next two processes in the cycle.

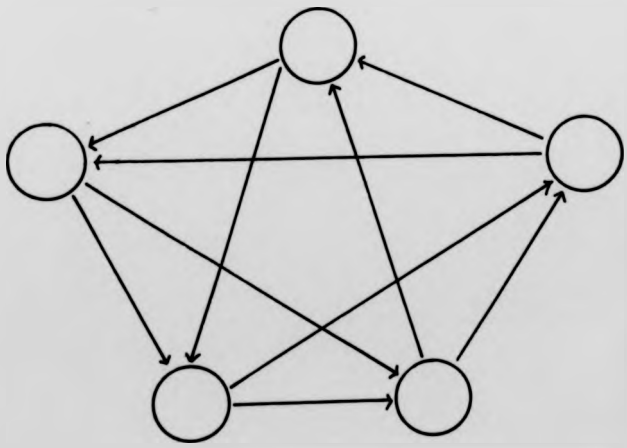


Figure 10b

Every computer has M modules of P processes, and each process has R inter-module output channels. Inter-module output channels from the pth process of the mth module will be connected to the pth process of the mth module on R different computers. The computers are organised into a circle, and a computer with M*P*R remote channels will be connected to the next M*P*R computers in the circle, excluding itself. For example, Figure 10c shows how the inter-module output channels of the four processes on the left-most computer would be connected in the configuration C=4, M=2, P=2 and R=1.

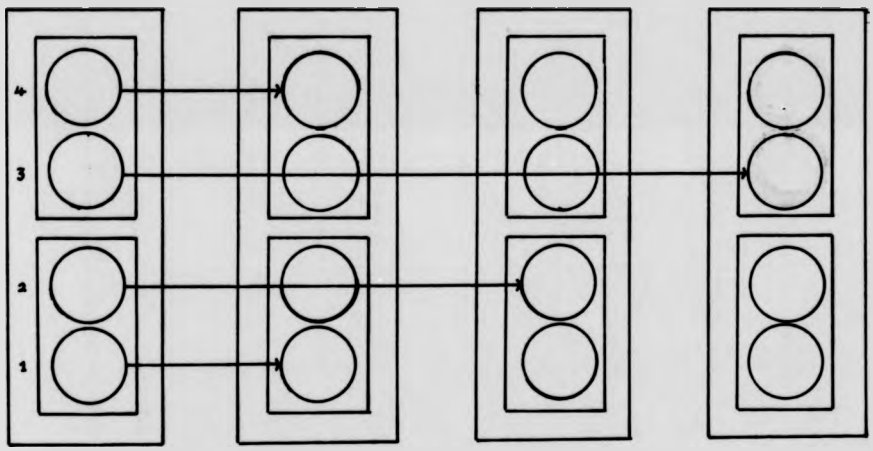


Figure 10c

All inter-module channels have been configured so as to be inter-computer. This means that the network loading is worst-case, as in a real system some inter-module channels would be intra-computer. One of our experiments is to investigate the advantages of reconfiguring the application level onto more computers, and by having all inter-module channels inter-computer we ensure that despite reconfiguration the network loading remains constant and so does not affect our results.

10.2.3 Distributed kernel level

The processes are time sliced. Each computer has a ready-queue. When a process is ready to run it is placed at the end of its host's ready-queue. The process at the front of the ready-queue is run for a time slice of 25 milliseconds or until it suspends itself on a SEND or a RECEIVE. Paging is assumed but it is not modelled.

Messages are transferred in basic blocks. However it is not sufficient to rely on the basic block protocol as it does not provide recovery from errors during transmission. To be able to recover from such errors a higher level protocol incorporating a positive acknowledgement is needed. This protocol is provided by the kernel.

Although we assume that all transfers are error free it is necessary to model the higher level protocol so that the network load is realistic. When a message is sent, it is placed on the appropriate queue in the sender's ALU. When it is received by the receiver's ALU it is placed in the receiver's input channel and an acknowledgement message is placed on the appropriate output queue in the receiver's ALU. Once the acknowledgement has been delivered to the sender's ALU the SEND is completed and the sender is activated and placed on the ready-queue again.

10.2.4 Process survivability

Every process has the same level of redundancy. For the module-based secure point strategy to work the backup replicates of processes in the same module must be on the same computer (see Chapter 9, Section 9.5). This organisation is also used when simulating the worst-case secure point insertion strategy even though it is not necessary. The backups are distributed fairly between the computers.

Two secure point insertion strategies are modelled: worst-case and module-based. When a process executes a secure point, secure point data is sent to each of the kernels hosting that process's backup replicates.

Secure point data consists of those pages (a one Kword page is used) of the process's data segment that have been altered since the last secure point plus a copy of the process's volatile environment. The number of pages that have been altered since the last secure point is defined by the bar chart shown in Figure 10d, where the number of pages is determined by the number of CPU seconds of work performed since the last secure point.

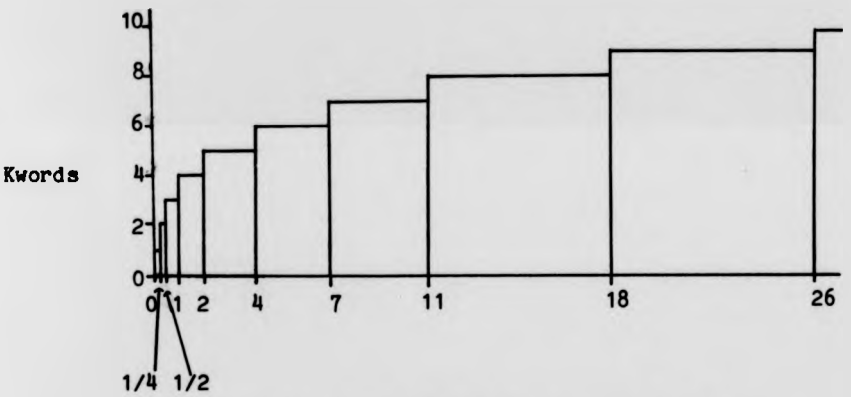


Figure 10d

The bar chart is based on the 'working set' curve (Denning 1968, Spirn 1977). The working set curve, first described by Denning, describes how the number of different pages accessed in time t (known as the working set) increases with t. The values along the axes are 'guesstimates' as there are no suitable published figures on which to base them.

The two stage protocol that is used to broadcast the secure point data is performed by the processes themselves. A basic block can contain at most 1K words and so secure point data is divided into a header block followed by a number of blocks each containing one page. The header contains the number of pages, details as to which pages they are, and a copy of the process's volatile environment. Each of these messages is sent separately by the user process.

Some of the secure point message's pages would be in main memory and others in secondary storage. The latter would increase the time it takes to perform a secure point. However as paging is not modelled, we assume that all pages are in main memory and hence ready to be sent immediately. For the same reason we do not model the updating of the backup replicates with the secure point data; in fact the backup replicates are not modelled at all.

10.3 Implementation details

The simulation is performed by two programs. One generates a PROSUR configuration, and the other is the discrete event simulator that simulates the running of that configuration.

The configuration generator takes as input the number of computers, the number of modules per computer, the number of processes per module, and the number of local and remote channels per process. It produces as output a file containing a description of the distributed computer control system.

The discrete event simulation takes as input the configuration, the level of redundancy, the type of process survivability to be modelled, and the mean work period of the processes.

Both programs are written in Pascal. The configuration generator is 158 lines long, and the simulator itself is 1498 lines long. Development and testing was carried out on a local GEC 4190, and the final production runs were done on a University of Manchester Regional Computer Centre CDC 7600.

10.4 The experiments performed

In pursuit of the aims described in Section 10.1 a total of 138 distinct experiments were performed.

The application level simulated consists of 48 modules each of six processes. Each process is connected by intra-module output channels to all of the other processes in its module, and each process also has one inter-module output channel connected in the manner described in sub-section 10.2.2.

Four configurations were formed by partitioning the modules equally onto 4, 8, 16 and 32 computers. A common proposal for distributed computer systems is that there should be but one process per computer. In order to investigate any possible advantage this may ensure for process survivability a fifth configuration consisting of 192 computers, each with one process, was also simulated. Each process in this configuration has 6 inter-module output channels and no intra-module output channels.

192 processes divided into modules of 6 processes were chosen because they could be evenly partitioned a number of ways. Also, for larger numbers of processes the CDC Pascal's heap storage became exhausted.

Each configuration was simulated in a number of different situations, where a particular situation is defined by a combination of three factors:

1) the level of redundancy;

2) the secure point insertion policy used;

3) the SEND frequency.

The combination of configuration and situation defines an experiment.

The 8, 16, 32 and 192 computer configurations were simulated with redundancy levels of 1, 2, 3, 4 and 6. The 4-computer configuration was simulated with redundancy levels of 1, 2 and 3.

Each configuration was simulated with every level of redundancy using both worst-case and module-based secure point insertion. Each configuration was also simulated without process survivability in order to measure its normal performance.

A process is modelled as a RECEIVE-work-SEND loop. The work period is Normally distributed with the same mean and standard deviation for all processes. To investigate the effects of increasing the SEND frequency the above experiments were repeated using a mean work period of 2, 0.5 and 0.12 seconds and standard deviations of 1, 0.1 and 0.01 seconds respectively.

Each simulation run simulates 30 minutes of real time. All of the measurements are reinitialised after 10 minutes in order to prevent the final results from being affected by any abnormalities produced while the simulation settles down. The final results are produced by the last 20 minutes.

10.5 The results

10.5.0 Introduction

The result produced by each experiment is the average of the useful work performed by each process. The results of all the experiments are presented in three sets of graphs - Figures 10e, 10f and 10g. These results are also given in tabulated form in Appendix D. The accuracy of these results is discussed in Appendix E.

Each set consists of two graphs. One graph presents the performance of all the configurations with worst-case secure point insertion and the other their performance with module-based secure point insertion.

Each set of graphs plots the configurations' performances with different mean work periods: Figure 10e - 2 seconds, Figure 10f - 0.5 seconds, and Figure 10g - 0.12 seconds.

A configuration's performance in CPU seconds is plotted against its level of redundancy. The points for a particular configuration are joined together to form a curve, and the curve is labelled with the number of computers in that configuration. The value plotted against a level of redundancy of zero is the configuration's performance without process survivability. Performance is the average of the amount of useful work performed by each process, and the standard deviations of the processes' performances are shown to scale as vertical bars.

These results are described in the following sub-sections. All comments are made with the proviso that they are only relevant to the configurations and situations simulated. Speculation concerning the performance of configurations and situations not simulated is left until the conclusion. The possible causes of the results described in this section are discussed in the following section, Section 10.6.

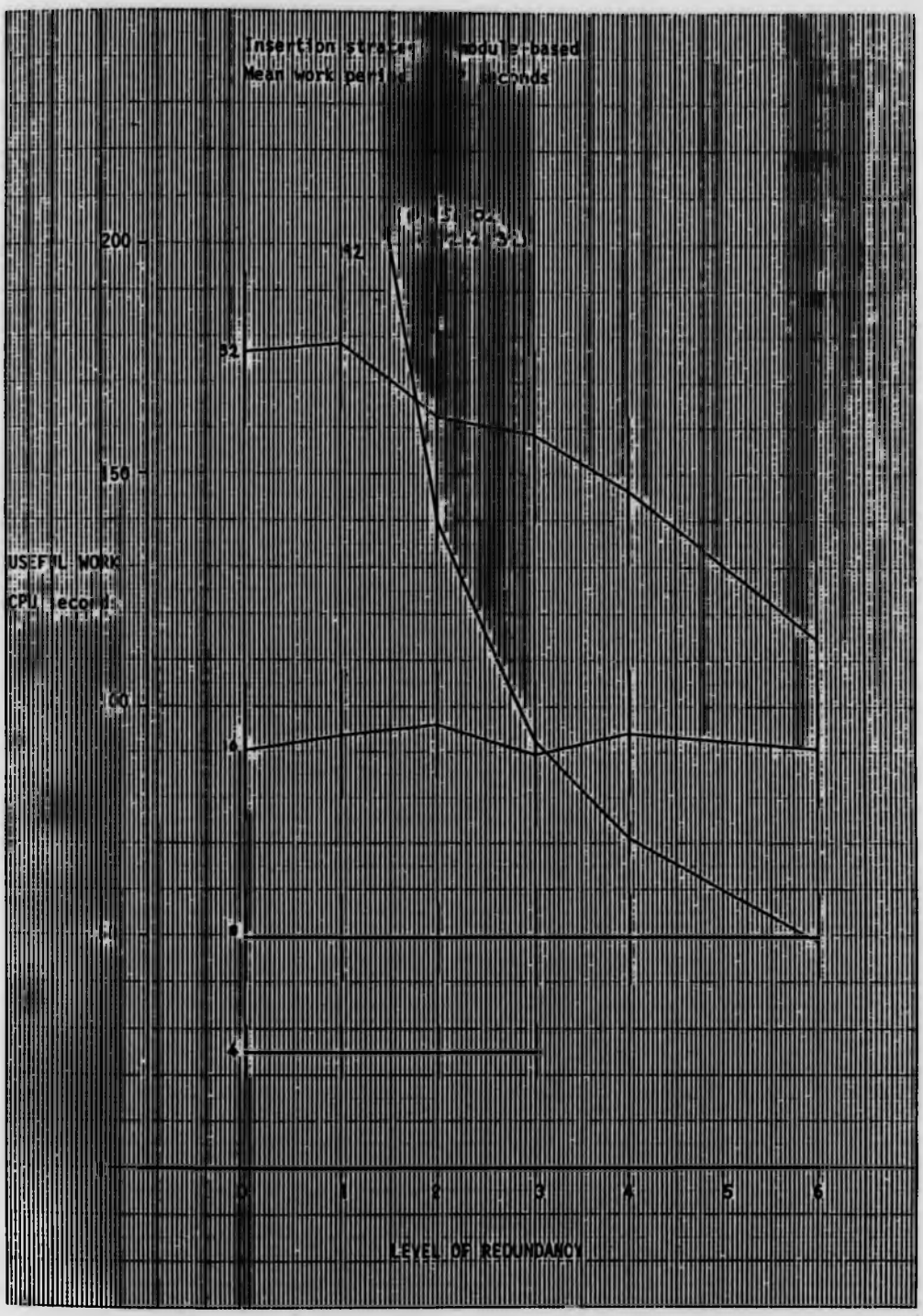

Insertion strategy: module-based
200
150
USEFUL WORK
CPU seconds
42
32
4
1
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 10e.1

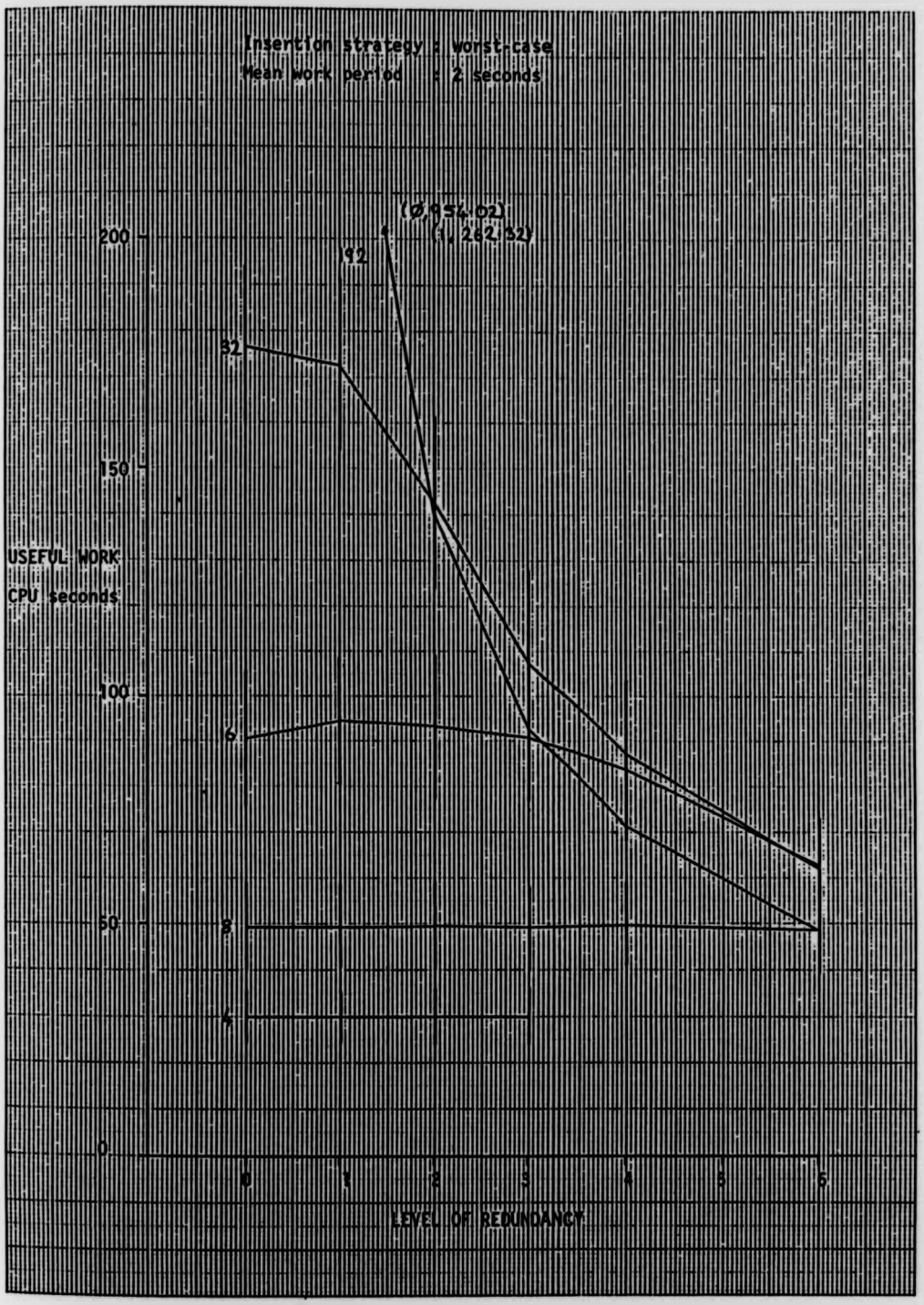

Insertion strategy : worst-case
Mean work period : 2 seconds
(0,954.02)
(1,262.32)
92
32
6
8
4
200
150
100
50
0
USEFUL WORK
CPU seconds
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 10e.2

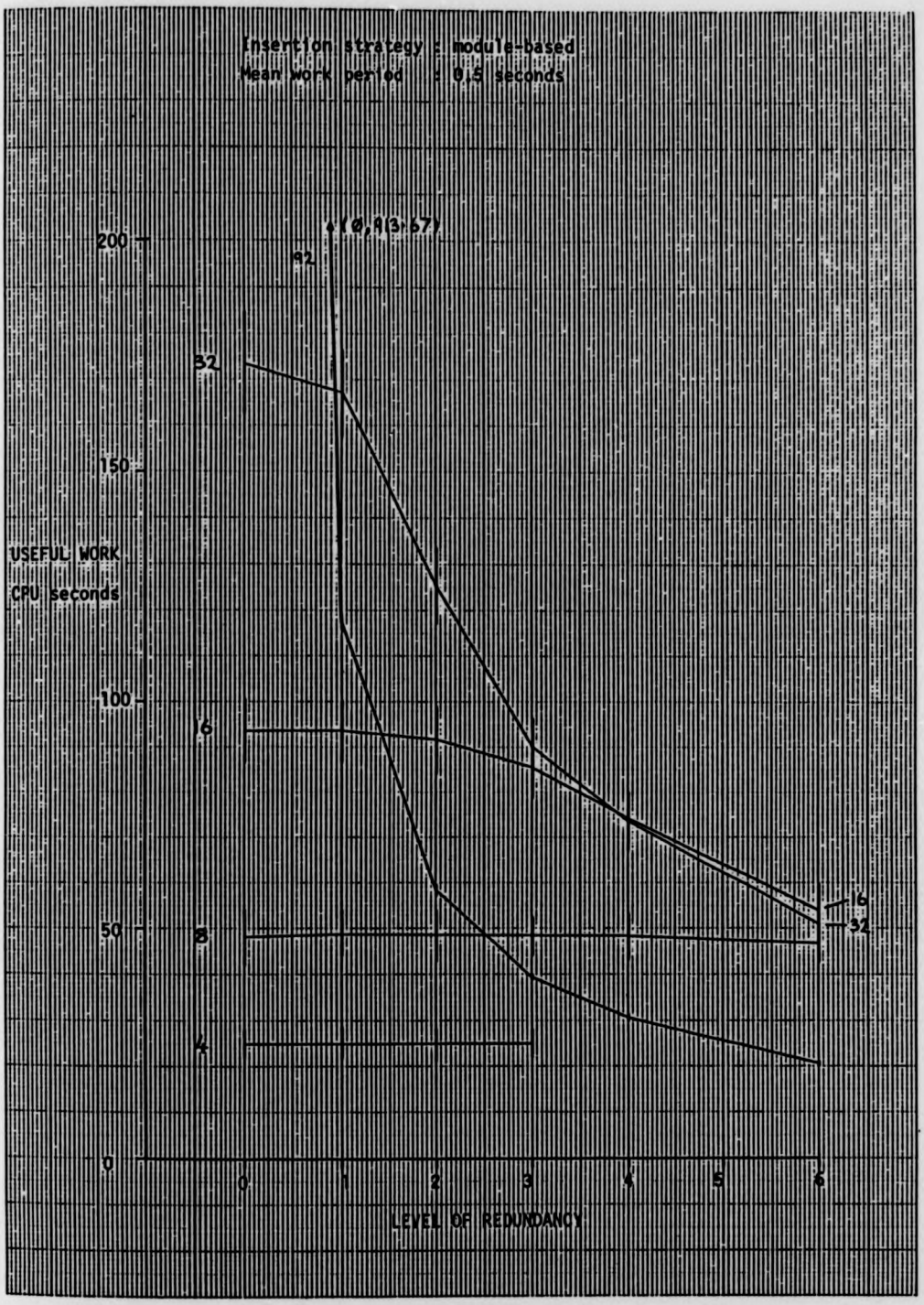

Insertion strategy : module-based
Mean work period : 0.5 seconds
(0,9(3.67)
200
150
100
50
0
USEFUL WORK
CPU seconds
32
16
8
4
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 10f.1

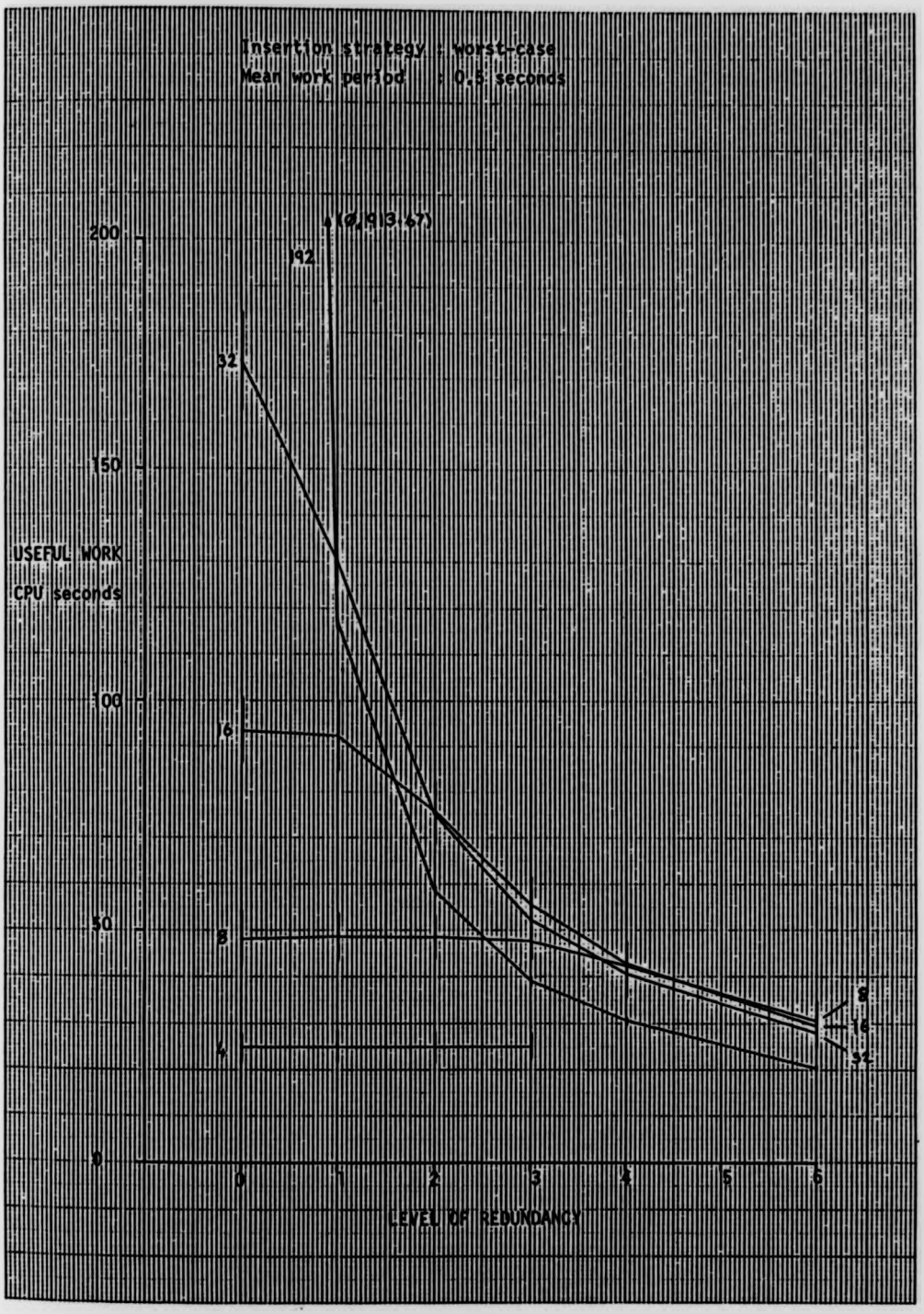

Insertion strategy : worst-case
Mean work period : 0.5 seconds
(0,9|3.67)
192
200
150
100
50
0
32
16
8
4
USEFUL WORK
CPU seconds
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY
8
16
32

Figure 10f.2

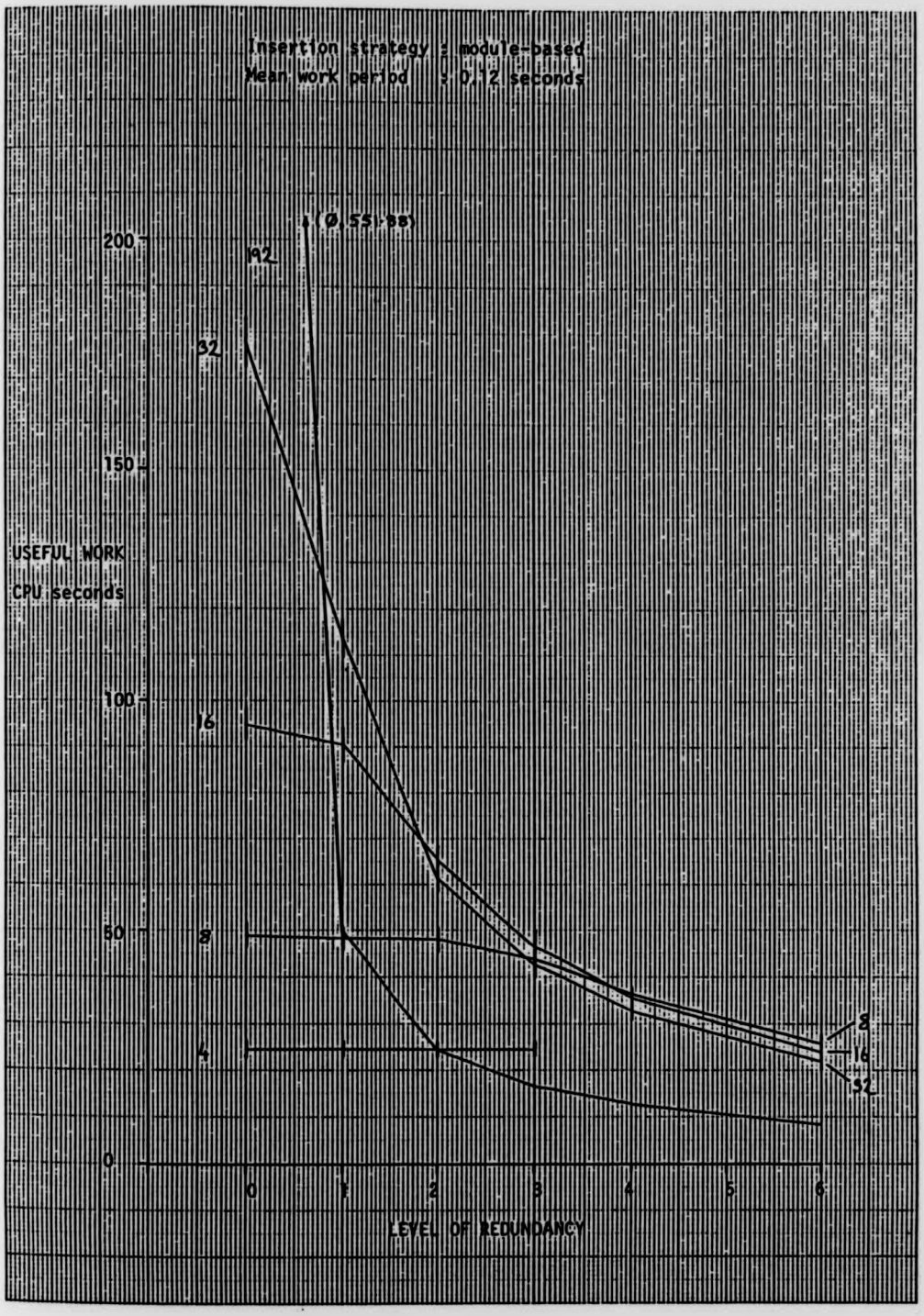

Insertion strategy : module-based
Mean work period : 0.12 seconds
1 (0.55-88)
92
32
16
8
4
200
150
100
50
0
USEFUL WORK
CPU seconds
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY
8
16
32

Figure 10g.1

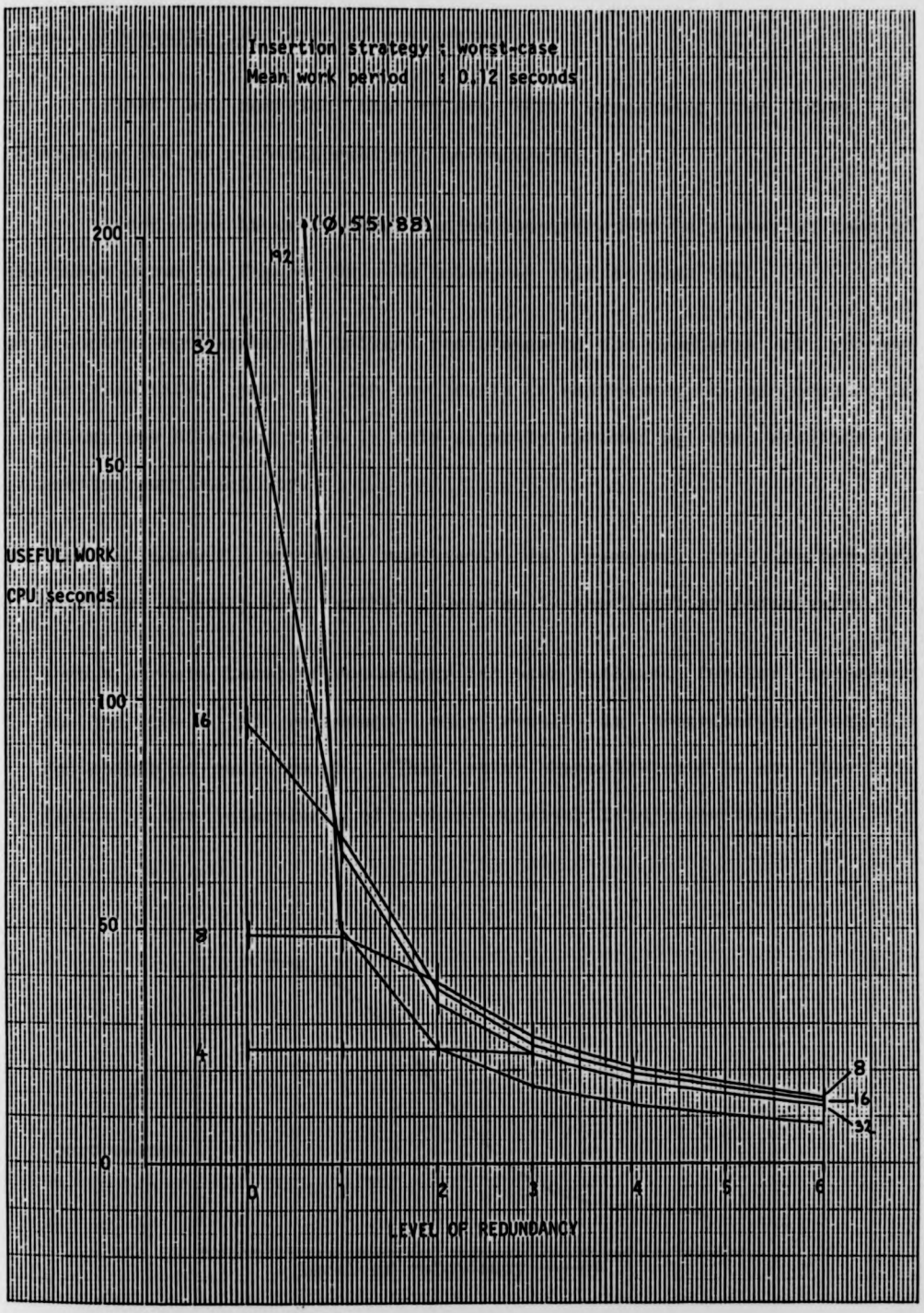

Insertion strategy : worst-case
Mean work period : 0.12 seconds
(0.55, 0.88)
200
150
100
50
0
USEFUL WORK
CPU seconds
32
16
8
4
8
16
32
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 10g.2

10.5.1 Increasing the level of redundancy

First we describe the effect that increasing the level of redundancy has on a configuration's performance.

Every line in every graph can be considered to be a special case of the curve shown in Figure 10h. We interpret this to mean that:

a) A control system has an initial resilience to the overheads of process survivability, and so for low levels of redundancy its performance is either not affected or it is only slightly reduced by the addition of process survivability.

b) Once a control system's resilience is overcome, its performance decreases sharply with further increases in the level of redundancy.

c) When a certain level of redundancy has been reached, further increases in the level of redundancy only result in progressively smaller reductions in performance. (This latter characteristic is an important factor in the determining of process survivability's practicality.)

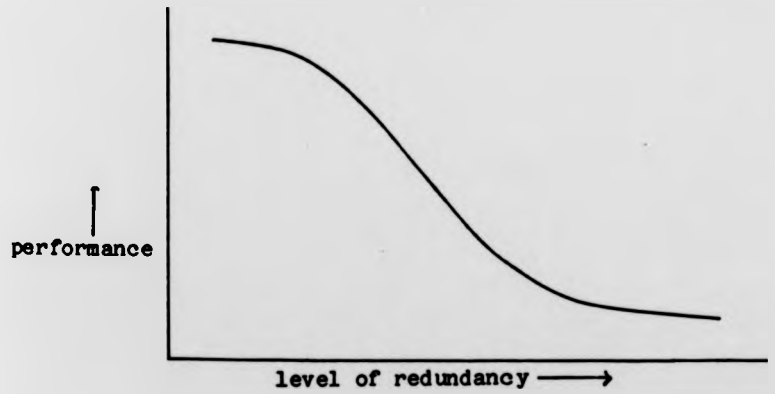


Figure 10h

The shape of a specific configuration's performance curve is determined by the balance of its natural resilience and the process survivability overheads of its situation. Some lines are virtually horizontal, for example lines 4 and 8 in Figure 10e.2, implying that in those configurations their resilience is not overcome and so their performance is maintained for all of the levels of redundancy tested. On the other hand, in some of the configurations resilience is immediately overcome by the overheads, for example in Figure 10f.2 line 32 falls sharply for low levels of redundancy.

10.5.2 Varying the process to computer ratio

Five configurations were simulated in order to study the effect that varying the process to computer ratio has on performance. In the following we distinguish between the configurations by their process to computer ratios.

Each of the six graphs plots the performance of all five configurations over the full range of redundancy levels with the same SEND frequency and with the same secure point insertion strategy. As the other factors are constant it is concluded that the differences between the configurations' performances must be due to their different process to computer ratios.

The following comments are based primarily on Figures 10f and 10g. The effects described below are not as well developed in Figure 10e, although a tendency towards them can be seen.

All comparisons are made between lines within the same graph, and not between lines in different graphs.

When simulated without process survivability the performance figures are approximately inversely proportional to the configurations' process to computer ratio. The lower the ratio the better the performance. Doubling the number of computers and halving the number of processes per computer results in an approximate doubling of the system's performance; a suitable return for the extra cost.

The most obvious difference between the configurations is that the smaller the process to computer ratio the greater the degradation in the configuration's performance. To be more precise, the smaller the ratio:

a) the lower the level of redundancy needed to overcome the configuration's resilience; and

b) the steeper the initial drop in performance.

As the lower ratio configurations are more adversely affected, their performances converge on the performances of the higher ratio configurations as the level of redundancy is increased. Convergence occurs in the order of increasing process to computer ratios; for example, in Figure 10f.2, line 192 converges on line 32, these two then converge on line 16, and finally all three converge on line 8. After two or more configurations' performances have converged, their performances remain converged for all further increases in the level of redundancy, and their performances slowly decline together as the level of redundancy increases.

Once converged, the performance of the lower ratio configuration(s) falls marginally below that of the higher ratio configuration(s). For example, in Figure 10f.2, when lines 192 and 32 converge on line 16, they both drop below line 16 and remain there. Performance is an average and so the performances of the individual processes in a configuration are spread over a range of values, with the size of the spread indicated by the standard deviation. A certain amount of overlapping between each

configuration's range of performances can be seen, making such a ranking of configuration performances less distinct.

Despite the configurations' widely different initial performances, the convergence means that for the higher levels of redundancy a similar performance can be achieved with 8 computers as with 16, 32 or 192 computers, a considerable financial saving.

As observed in the previous sub-section, some configurations' performances are unaffected by the introduction of process survivability no matter what level of redundancy is used. These unaffected configurations are always those with the higher ratio of processes to computers.

10.5.3 The two secure point insertion policies

Two different secure point insertion strategies were simulated. The worst-case insertion strategy results in a secure point being performed every SEND. The module-based insertion strategy results in less secure points but the secure point messages are larger.

Figure 10i consists of three graphs - 10i.1, 10i.2 and 10i.3 - which were formed by superimposing the two graphs in each of Figures 10e, 10f and 10g respectively. (Standard deviations are omitted.) Each configuration's performance curve is labeled by the number of computers in that configuration, and by the letter 'w' or 'm' to indicate whether the performance curve was obtained using worst-case or module-based secure point insertion. These three graphs show the relative effects of the two insertion policies on each configuration for each SEND frequency simulated. Again comparisons are not made between performances obtained with different SEND frequencies.

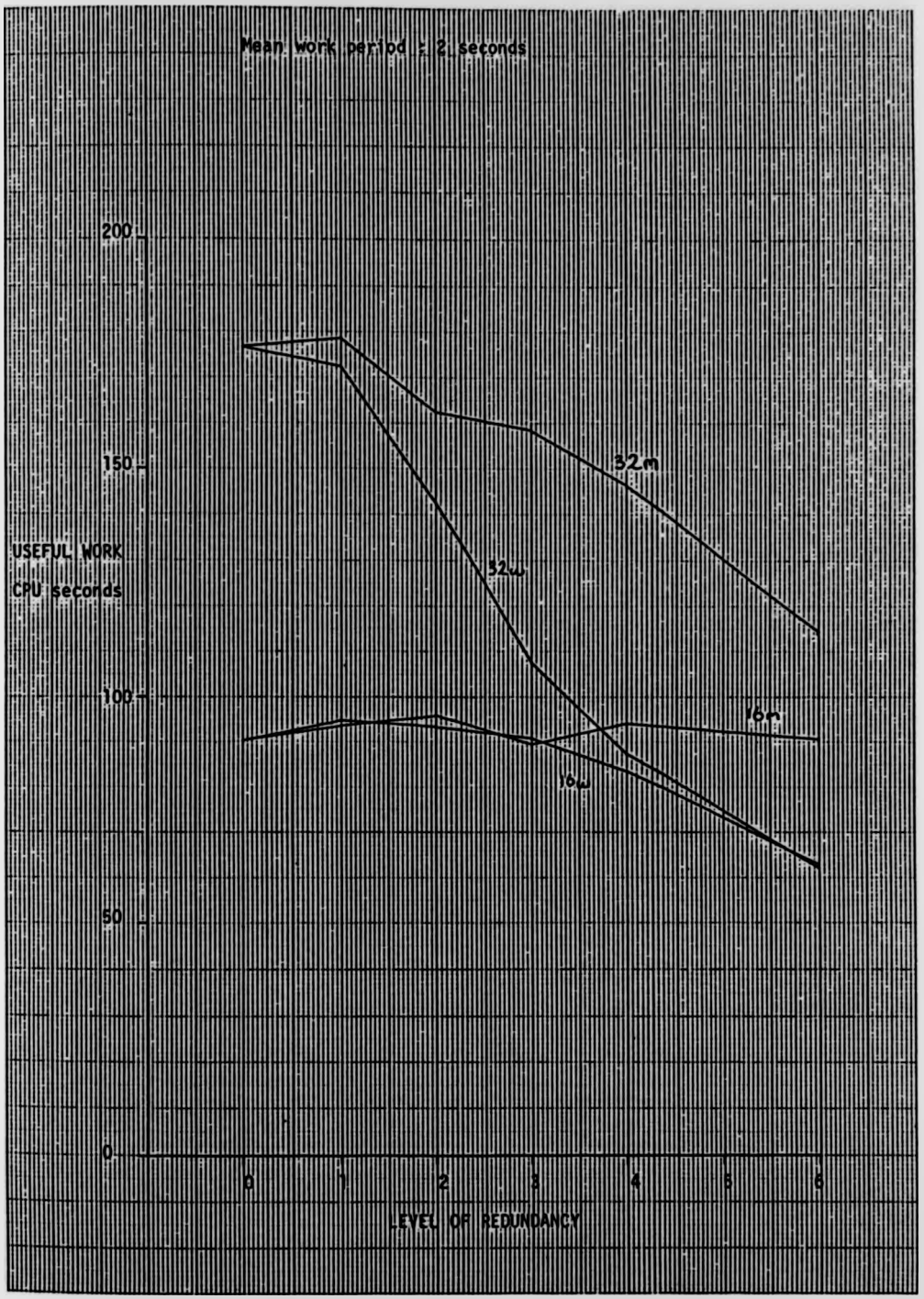

Mean work period = 2 seconds
200
150
100
50
0
USEFUL WORK
CPU seconds
32m
32w
16m
16w
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 10i.1

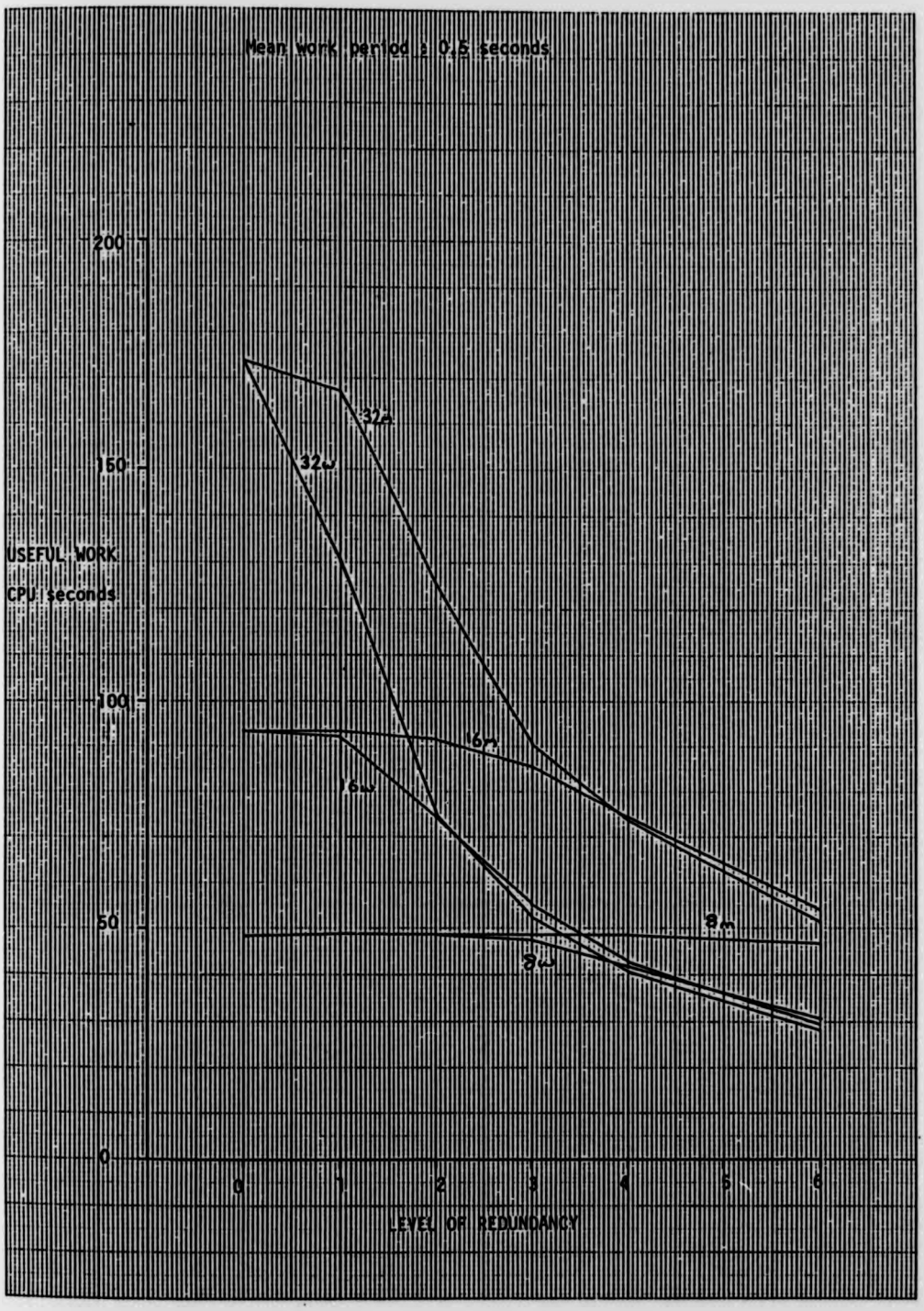

Mean work period = 0.5 seconds
200
150
100
50
0
USEFUL WORK
CPU seconds
32m
32w
16m
16w
8m
8w
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 101.2

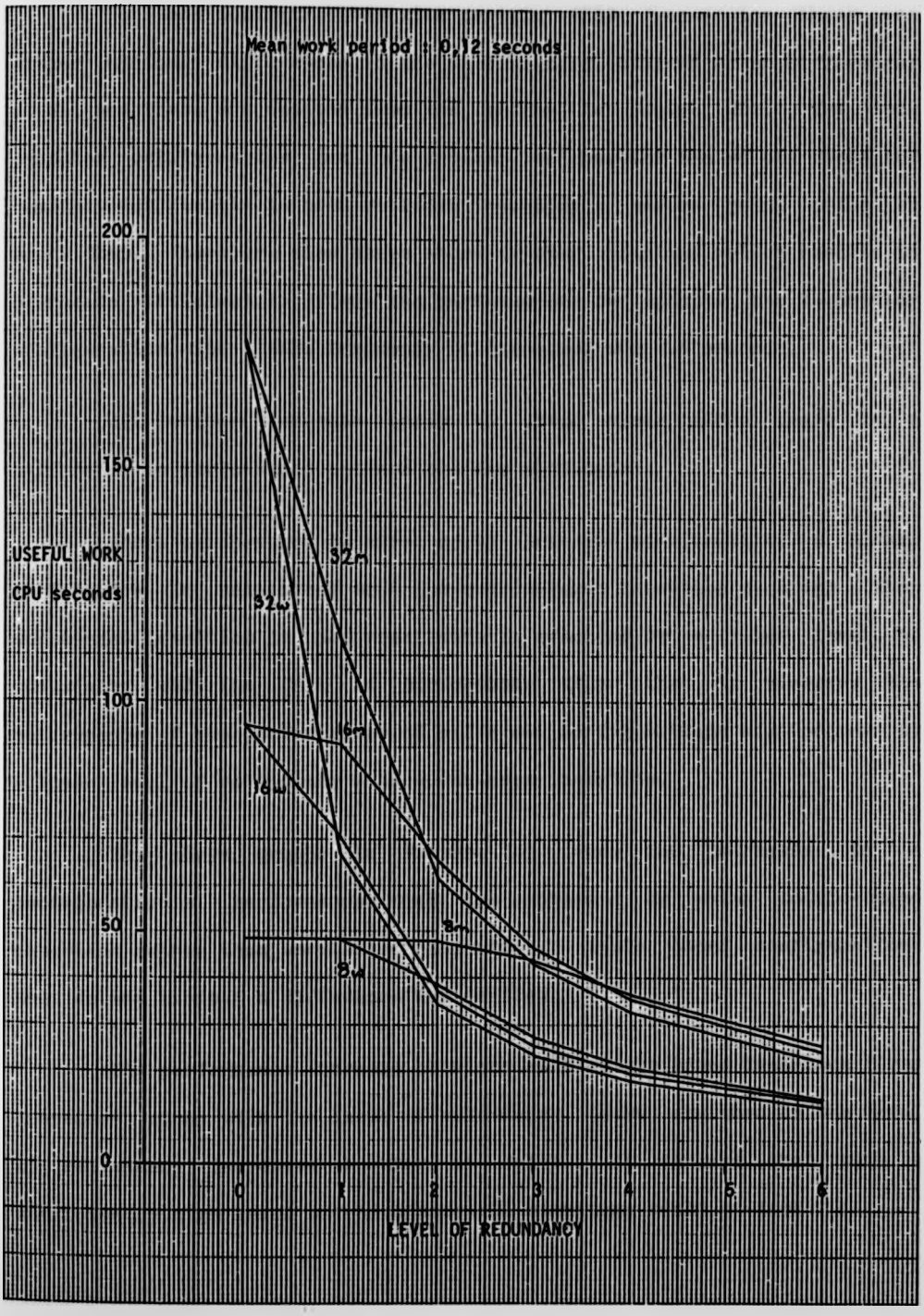

Mean work period : 0,12 seconds
200
150
100
50
0
USEFUL WORK
CPU seconds
32
32
16
16
8
8
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 101.3

For some configurations with certain SEND frequencies there are either minimal differences in the performances obtained or no differences at all and these are omitted from the graphs in Figure 10i. The 192-computer configuration has one process per computer and so all SENDs are remote which results in both policies inserting a secure point prior to every SEND. There are also cases where neither insertion policy results in overheads sufficient to affect the performance; for example, the 4-computer configuration with a SEND frequency of 2 seconds and 0.5 seconds.

All further comments apply to those cases where there is a difference in performance between the two strategies.

Where performance is affected by process survivability a higher performance is maintained with the module-based insertion strategy. When worst-case insertion is used resilience is overcome by a lower level of redundancy and the fall in performance after that is steeper. The convergence of the performances of different configurations also occurs at a lower level of redundancy with worst-case insertion.

The three graphs in Figure 10j show the increases in a configuration's performance that are achieved by using module-based insertion rather than worst-case insertion. Again those configurations that do not exhibit a difference are omitted.

The performance increase obtained with module-based insertion first increases along with the level of redundancy, peaks, and then as the level of redundancy increases further it falls. It appears that with a sufficiently high level of redundancy all advantage would be lost.

For some configurations with certain SEND frequencies there are either minimal differences in the performances obtained or no differences at all and these are omitted from the graphs in Figure 10i. The 192-computer configuration has one process per computer and so all SENDs are remote which results in both policies inserting a secure point prior to every SEND. There are also cases where neither insertion policy results in overheads sufficient to affect the performance; for example, the 4-computer configuration with a SEND frequency of 2 seconds and 0.5 seconds.

All further comments apply to those cases where there is a difference in performance between the two strategies.

Where performance is affected by process survivability a higher performance is maintained with the module-based insertion strategy. When worst-case insertion is used resilience is overcome by a lower level of redundancy and the fall in performance after that is steeper. The convergence of the performances of different configurations also occurs at a lower level of redundancy with worst-case insertion.

The three graphs in Figure 10j show the increases in a configuration's performance that are achieved by using module-based insertion rather than worst-case insertion. Again those configurations that do not exhibit a difference are omitted.

The performance increase obtained with module-based insertion first increases along with the level of redundancy, peaks, and then as the level of redundancy increases further it falls. It appears that with a sufficiently high level of redundancy all advantage would be lost.

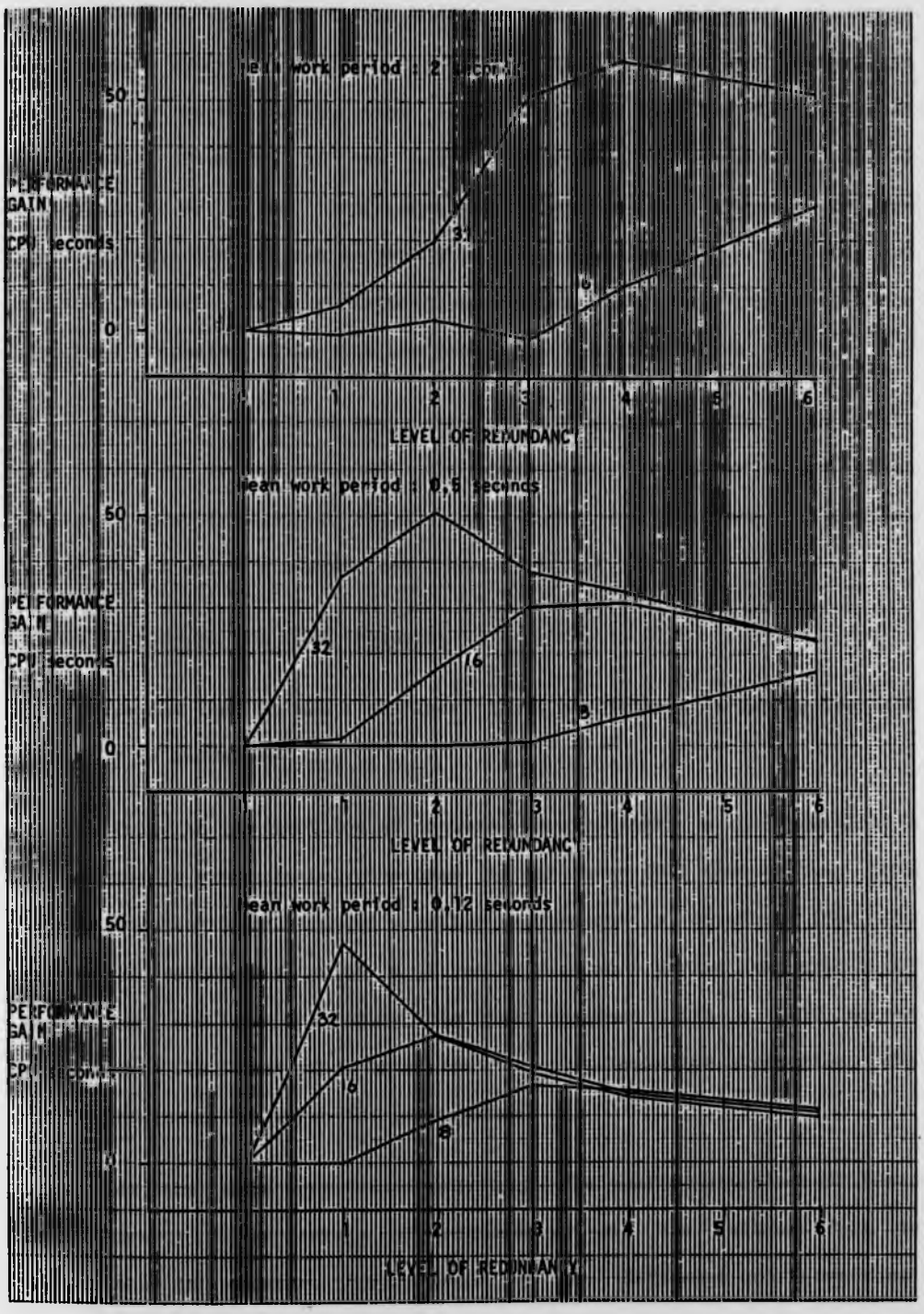

50
PERFORMANCE GAIN
CPU seconds
0
32
16
8
1
2
3
4
5
6
LEVEL OF REDUNDANCY
Mean work period : 0,5 seconds
Mean work period : 0,12 seconds

Figure 10j

Because of the convergence described in the previous sub-section which occurs with both insertion policies, the increase in performance obtained in each configuration is virtually the same for high levels of redundancy. However, if the extra performance is considered as a percentage of the configuration's performance without process survivability, then it can be seen that the greater advantage is obtained in the highest ratio configuration; viewed this way the performance advantage decreases along with the process to computer ratios.

These observations imply that the overheads are higher with worst-case insertion, as was expected. The transmission of smaller secure point messages more frequently results in a higher overhead than does sending larger secure point messages less frequently.

10.5.4 Increasing the SEND frequency

We now look at how a configuration's performance varies as its SEND frequency is increased.

Prior to the addition of process survivability a configuration's performance is approximately the same for all three SEND frequencies. Once process survivability is added a better performance is obtained (for all levels of redundancy) with a 2 second mean work period than with a 0.5 second mean, and both are better than that obtained with a 0.12 second mean. This is true for both secure point insertion strategies. The smaller the mean the worse the performance.

As the mean work period decreases, the configuration's resilience is overcome by lower levels of redundancy, and the initial fall in performance is sharper. Convergence of the different configurations' performances occurs at a lower level of redundancy.

These results imply that overheads are greater with the faster frequencies. More frequent SENDs result in more frequent secure points which, although the secure point messages will be smaller, result in greater overheads.

10.6 The causes of the effects described

10.6.0 Introduction

In this section we present possible explanations for the effects that were described in the previous section.

10.6.1 Increasing the level of redundancy

In Section 10.5.1 we presented Figure 10h as being the general shape of a configuration's performance curve as redundancy increases. In this sub-section we explain what makes the curve that shape, and why all the performance curves in Figures 10e, 10f and 10g, even the horizontal ones, are special cases of this curve.

In order to illustrate our explanation we look at all five configurations when operating with worst-case secure point insertion and a mean work period of 0.5 seconds. The performance figures for these configurations when operating in these situations are presented in Figure 10f.2. (Any of the other combinations of insertion policy and mean work periods could have been used.)

All of the processes execute a RECEIVE-work-SEND cycle, where the SEND may be preceded by a secure point. The more of these cycles that a process performs the more work it performs.

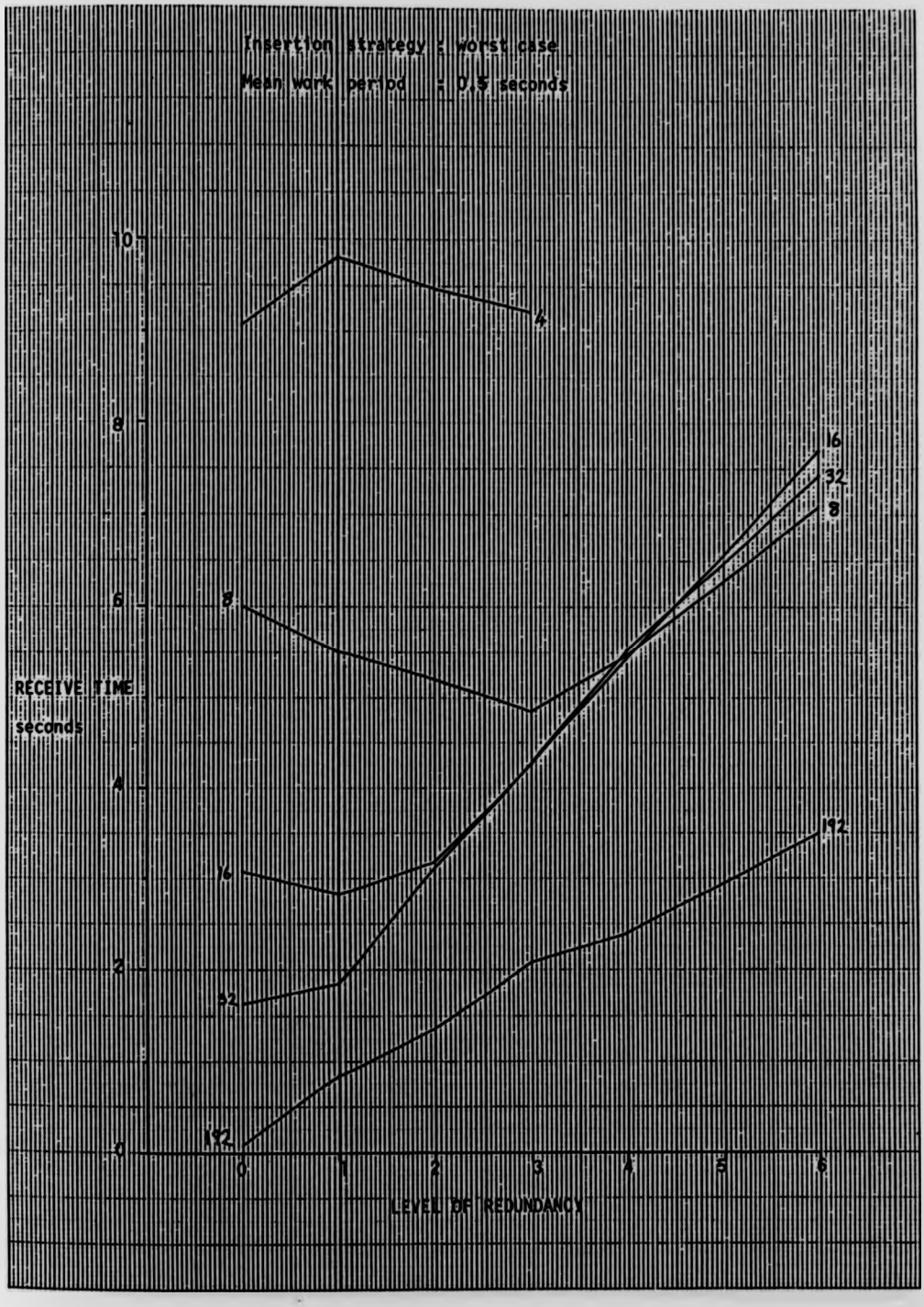

Insertion strategy : worst case
Mean work period : 0.5 seconds
RECEIVE TIME
seconds
LEVEL OF REDUNDANCY
0
2
4
6
8
10
0
1
2
3
4
5
6
4
8
16
32
64
128
128
64
32
16
8

Figure 10k

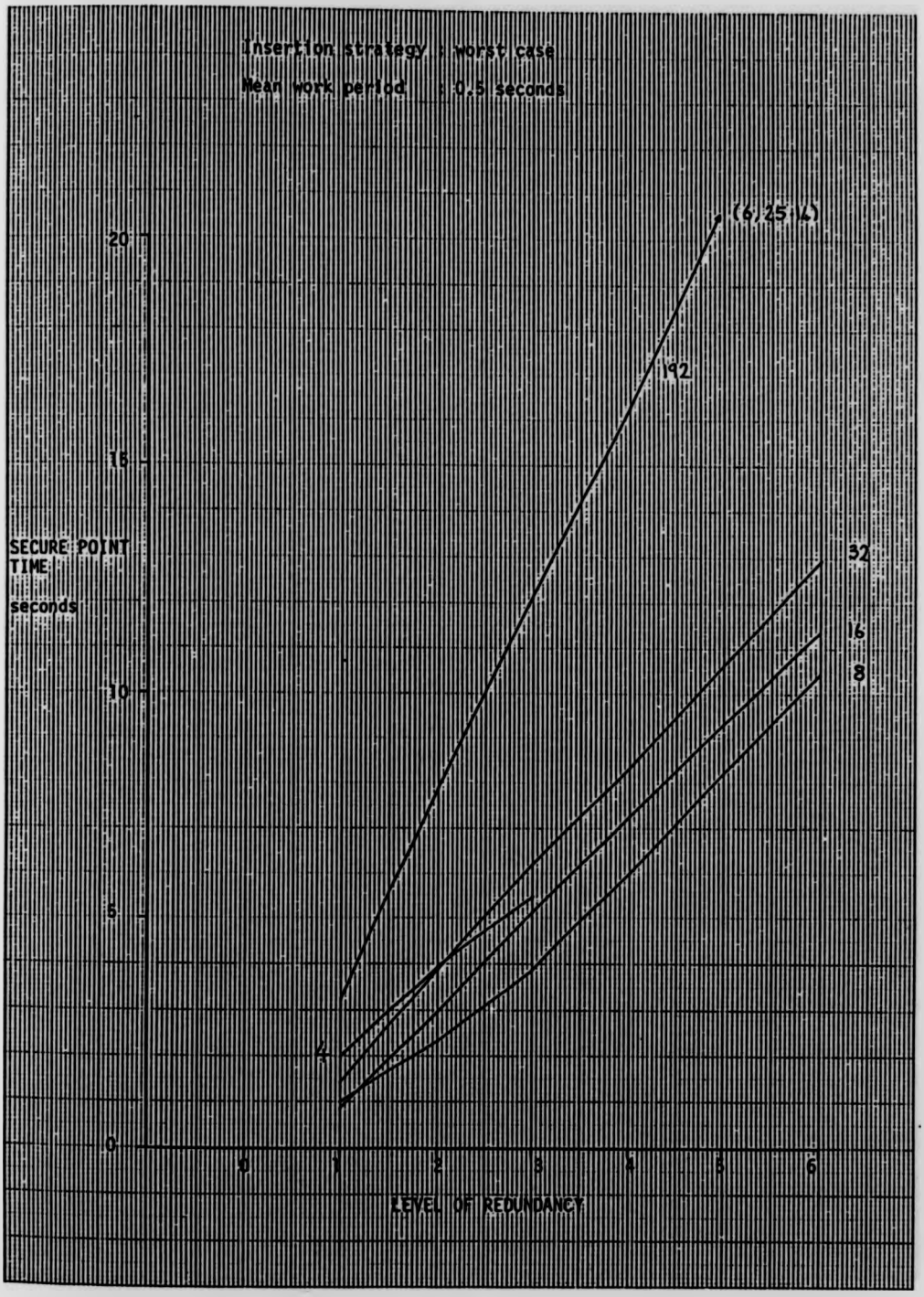

Insertion strategy : worst case
Mean work period : 0.5 seconds
(6,25.4)
192
32
16
8
4
20
15
10
5
0
SECURE POINT TIME
seconds
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY

Figure 101

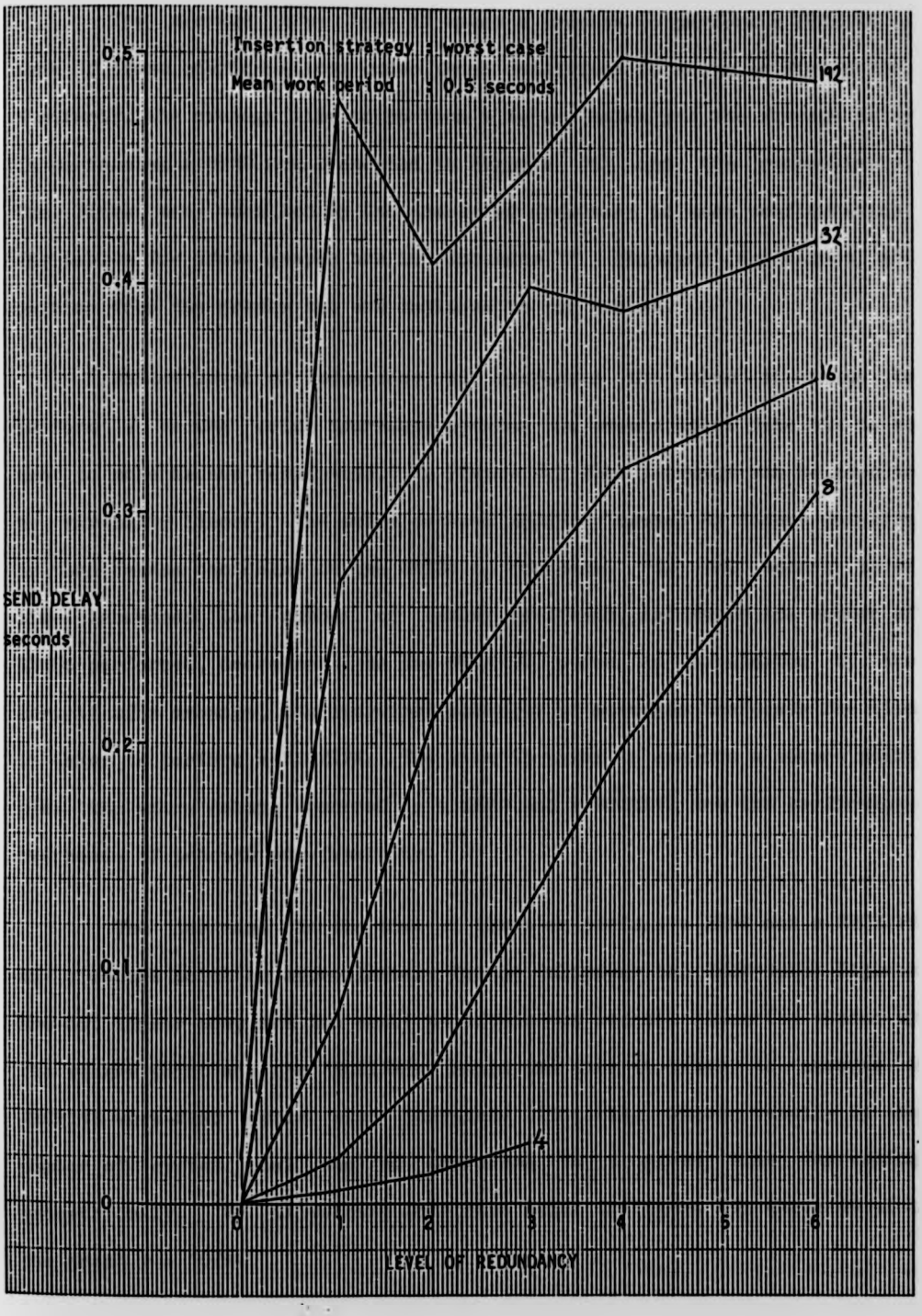

Insertion strategy : worst case
Mean work period : 0.5 seconds
0.5
0.4
0.3
0.2
0.1
0
SEND DELAY
seconds
0
1
2
3
4
5
6
LEVEL OF REDUNDANCY
128
32
16
8
4

Figure 10m

The average time that it takes a process to perform a RECEIVE in each of these configurations under these conditions is shown in Figure 10k. Similarly, the average time to perform a secure point and the average delay in performing a SEND are presented in Figures 10l and 10m respectively. The lines are labelled by the number of computers in each configuration. The tables for these graphs are in Appendix F. Standard deviations are omitted from the graphs but are included in the tables.

As the level of redundancy is increased the time it takes to perform a secure point increases because the secure point data has to be sent to more backup replicates. In Figure 10l this increase can be seen to be virtually linear. The extra network activity caused by secure pointing in turn causes the delay on the SENDs to increase. As the secure point time increases the number of cycles performed, and hence the number of SENDs performed, decreases. As the RECEIVE phase is terminated by the arrival of a message, the increased scarcity of messages increases the time it takes to perform a RECEIVE. The time it takes to execute a secure point plus the extra delays incurred in a SEND and a RECEIVE are the performance overheads of process survivability.

The increase in the time it takes to perform a SEND and a RECEIVE and a secure point is approximately linear (although the SEND delay figures are not linear they are too small to affect the overall linearity caused by the secure point and the RECEIVE). Hence the time it takes to execute a RECEIVE-work-SEND cycle also increases linearly, which results in the number of cycles that can be performed in a given time decreasing geometrically. We illustrate this last point in Figure 10n where we plot the graph $y = 10/x$.

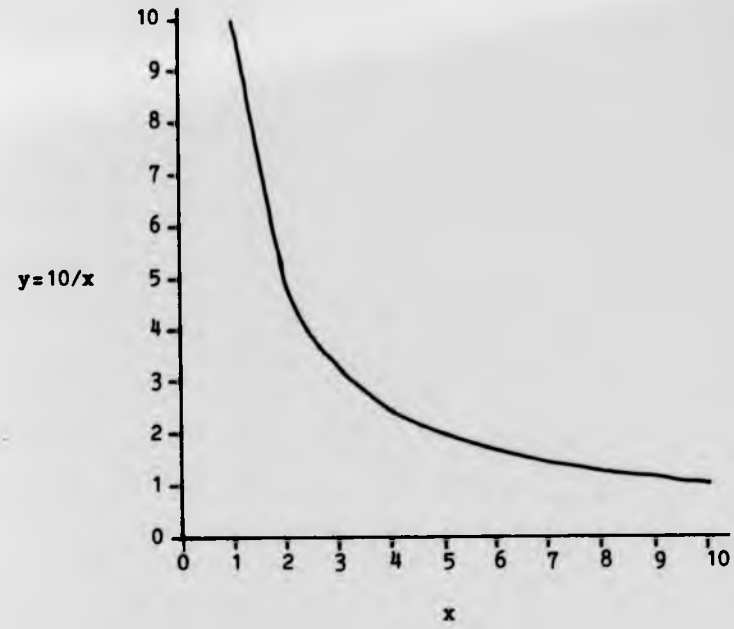


Figure 10n

We believe that the plunge and then the levelling-off of the performance curve shown in Figure 10h is due to the nearly linear increase in the time it takes to perform a secure point and a SEND and a RECEIVE.

We turn now to the cause of the initial resilience that a configuration's performance has to the process survivability overheads, and answer the question of why the curve in Figure 10h is initially horizontal.

Processes on the same computer are time shared. Those processes that are executing the work phase of a RECEIVE-work-SEND cycle are 'contained' in the computer's ready-queue. The more processes there are in a computer's ready-queue the longer it will take those processes to complete their work phases.

We believe that a configuration's initial resilience is due to a combination of two factors:

1) While the number of processes in the ready-queue is high, and the process survivability overheads are low, the predominant factor in determining a process's performance will be the time that it takes the process to perform its work phases. As the process survivability overheads increase, the ready-queue length will decrease and the predominant factor will become the time it takes to perform the SENDs, RECEIVEs and secure points, and performance will fall.

2) As the process survivability overheads increase the number of processes that are executable at the same time decreases and so does the number of processes on the ready-queue. This reduction means that processes can perform their work phases faster, which partly compensates for the increase in the time spent RECEIVEing, secure pointing and SENDing.

Hence, a configuration's performance is limited for small levels of redundancy by the length of its ready-queue, and then for the higher levels of redundancy by the process survivability overheads. As the process survivability overheads increase the ready-queues' sizes decrease, and initially it is possible for this to compensate for the increased process survivability overheads, thereby maintaining the processes' cycle speed and performance. Once the process survivability overheads have reached a certain level (and this varies between configurations and situations) the reduction in ready-queue sizes no longer compensates for the increased overheads and so performance falls. The rate at which performance falls is still alleviated by the shorter ready-queues.

Not all the curves in Figures 10e, 10f and 10g are the same shape as the general curve shown in Figure 10h. This is due to the different balances of ready-queue length and process survivability overheads in the

different configurations and situations.

It was noted in Section 10.5.1 that those configurations with a higher process to computer ratio have the higher resilience to the overheads of process survivability. In fact, in the case of the 4-computer configuration its performance curves are all basically horizontal. This resilience is due to these configurations having the longer ready-queues. Those configurations with a lower ratio of processes to computers, such as the 32-computer configuration, have shorter ready-queues and so the ready-queue length does not dominate the process survivability overheads to the same extent; hence the lower-ratio configurations have a lower resilience.

Prior to performing the simulation it was expected that the network would be the bottle neck and that system performance would be limited by the need to share the network. As we have explained above, performance is limited by other factors, and not by the network. As an example, Figure 10o presents the ALU performance for all five configurations, again operating under worst-case secure point insertion and with a mean work period of 0.5 seconds. (These results are also presented in tabular form in Appendix G.) ALU performance is the average number of seconds worked by each ALU presented as a percentage of the 20 minutes simulated.

In general, ALU performance increases linearly for the lower levels of redundancy and then levels off over the higher levels of redundancy. The level of redundance at which ALU performance levels off varies between the configurations and in the 4-computer configuration it does not occur at all.

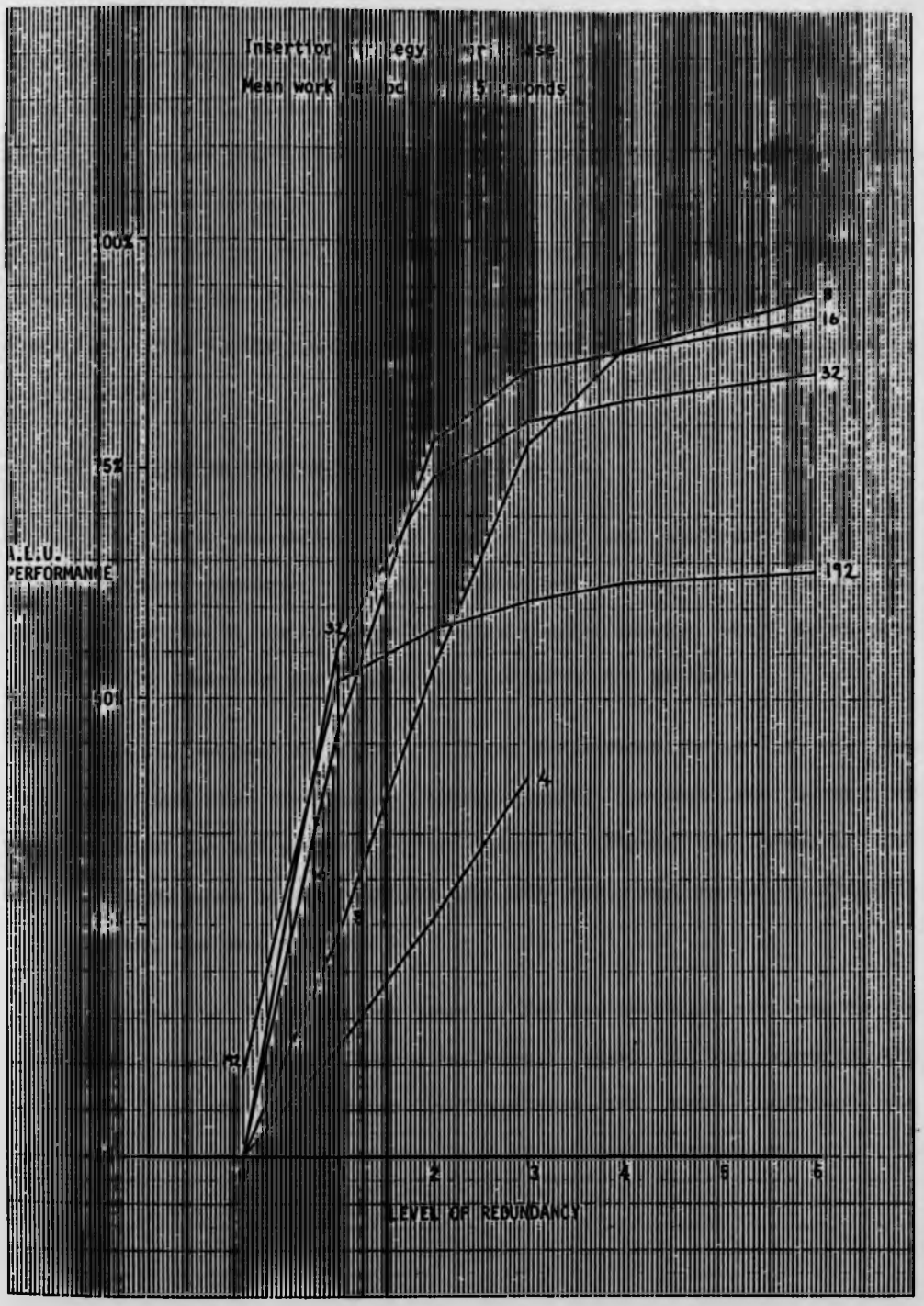


Figure 10o

Were the network to be the dominant factor we would expect all of the curves to approach 100%, and not level off below that. Also, referring back to Figure 10l, if the network were the limiting factor we would not expect the secure point times to increase linearly; neither would we expect the SEND delays, shown in Figure 10m, to level off as they do. We believe that the levelling off in ALU performance is due to a reduction in the demands made on the ALUs by the processes as it corresponds to the levelling off in the configurations' performance curves as can be seen in Figure 10f.2.

10.6.2 Varying the process to computer ratio

In this sub-section we explain why the performances of the configurations converge. To illustrate our discussion we use the 16-computer and the 32-computer configurations again running with worst-case secure point insertion and a mean work period of 0.5 seconds.

As can be seen in Figure 10f.2 the performance curves of the 16-computer and the 32-computer configurations converge for levels of redundancy of two and greater. The table in Figure 10p presents the average time spent in each RECEIVE-work-SEND cycle performing the RECEIVE, the SEND and the secure point.

Prior to a redundancy level of two the overheads are less in the 32-computer configuration. From a redundancy level of two onwards the overheads in the 32-computer configuration are worse, although the difference remains constant as the level of redundancy increases.

		configuration	
		16	32
level of redundancy	0	3.07	1.63
	1	3.77	3.53
	2	6.24	7.29
	3	9.65	10.88
	4	13.02	14.24
	6	19.44	20.78

Figure 10p

Despite the convergence in the process survivability overheads the 32-computer configuration should still have a better performance as there are less processes per computer, but as we can see in Figure 10f.2 this is not the case. The ready-queues in the 16-computer configuration will decrease in length as the redundancy level increases and at a certain level, two in this case, the length of its ready-queues will almost be the same length as in the 32-computer configuration and the processes will be able to execute their work phases as fast. For a redundancy level of 2 and greater, the length of the ready-queue in both configurations will be about the same size.

By halving the queue length and doubling the time it takes to send a basic block, the time it takes from request to completion to transmit a message, or to send secure point data, will remain about the same. This is why the process survivability overheads (which are predominantly made up of network utilisation) for the two configurations presented in Figure 10p are almost the same.

As we have explained before, the ALUs and the network are under utilised. Were this not the case, then the above would probably not be true.

10.6.3 Insertion policies and SEND frequencies

It was expected that worst-case secure point insertion would have a worse effect on the configurations' performances than module-based insertion would, and the results presented earlier support this assumption.

Figure 10q shows the average of the total time spent performing secure points by each process in the 16-computer configuration, operating with a mean work period of 0.5 seconds and with both secure point insertion policies. It can be seen that for all levels of redundancy a process will spend less time secure pointing with module-based insertion than with worst-case insertion.

		worst-case	module-based
level of redundancy	1	152.10	59.10
	2	425.96	154.32
	3	565.91	290.38
	4	624.82	389.40
	6	675.02	482.16

Figure 10q

Similarly, it was expected that increasing the SEND frequency would increase the number of secure points and decrease performance. That the performance is decreased has already been shown. Again using the 16-computer configuration operating with worst-case insertion, we show in Figure 10r that the total time spent by each process in secure pointing does increase as the SEND frequency increases.

level of redundancy	Mean work period: 2	0.5	0.12
1	54.90	152.10	476.93
2	126.25	425.96	642.96
3	238.02	565.91	683.84
4	382.15	624.82	697.33
6	526.99	675.02	721.19

Figure 10r

Worst-case insertion results in secure points being executed more often, and increasing the SEND frequency also increases the secure pointing frequency. These results are consistent with the graph, presented in Figure 10d, that defines the size of the secure point data, as there it can be seen that the higher the frequency of secure pointing the more data that must be transferred, as increasing the time since the last secure point does not increase the size of the secure point data proportionally.

10.7 Summary

Our experiments indicate that a high ratio of processes to computers is more practical for high levels of redundancy than a low ratio. For high levels of redundancy a comparable performance can be achieved with an 8-computer configuration as with 16, 32 or 192 computer configurations - a considerable financial saving.

The maximum level of redundancy that can be achieved in a distributed computer control system is limited by the number of computers. Because of this the larger computer configuration, although financially wasteful, may still be adopted in order to achieve the high levels of

redundancy required.

The attainment of high levels of redundancy is facilitated by the levelling off of performances for high levels of redundancy. If, for example, the performance obtained with a level of redundancy of six is acceptable, then the performance obtained with a level of eight is also likely to be acceptable.

Unfortunately if performance falls below an acceptable level then there is no way of improving it other than by adopting module-based secure point insertion. It was hoped that increasing the number of computers would increase the performance, but convergence prevents this.

The adoption of modules into PROSUR's P/L and the use of module-based secure point insertion leads to an improved performance over all levels of redundancy. The performance improvement is more significant in the higher ratio configurations where it forms a higher percentage of the performance that can be obtained without process survivability. As the level of redundancy is increased this advantage is reduced and it is possible that it might become negligible for very high levels of redundancy.

As expected the process survivability overheads increase as the SEND frequency increases. This may inhibit the adoption of process survivability for applications that involve a high frequency of message passing.

Response time in a real time system is a very important characteristic. Unfortunately it was not possible to measure response time directly. However, it is possible to hazard a guess as to how response time might be affected.

The most primitive response to a request is the RECEIVE-work-SEND cycle executed by every process. More complex requests would involve the server process issuing requests of its own to other processes. Hence, the increase in the time it takes to perform a RECEIVE-work-SEND cycle is a rough indication of the way that response time would be affected.

Referring back to the arguments presented in Section 10.6.1, the time it takes to perform the RECEIVE, SEND and secure point part of the cycle will increase linearly with the increase in redundancy. The time taken to perform the work phase of the cycle will depend on the size of the ready-queues and these will decrease as the time it takes to perform the rest of the cycle increases.

We believe that while performance remains unaffected by process survivability the response time will also be unaffected. Once the performance starts to fall, response time will increase linearly. As the ready-queues get smaller, there may be a slight decrease in the rate of increase but nothing as drastic as the levelling off that can be seen in the performance curves.

It was found that, contrary to expectations, configuration performance was not limited by the network. However, were the number of application processes to be increased, then the extra load on the network might result in the network becoming the limiting factor, and all of the above conclusions might be nullified.

Whether or not a particular performance or a particular response time is acceptable depends on the particular application involved. The most encouraging result is the levelling off of performance as the level of redundancy increases, rather than a steady fall. Unfortunately, this advantage may be nullified by the linear, or near linear, increase in response time as redundancy increases.

11. Conclusion

11.1 The aim of process survivability

Possibly the greatest advantage that a distributed computer control system has over a centralised system is that the failure of one or more of its constituent computers does not prevent the other computers from operating normally. This advantage can be further enhanced by basing the distributed computer control system around a local area network so that the computers can be physically dispersed over a large area thereby limiting the damage that could be caused by a major catastrophe such as a fire. Unfortunately the loss of the executive and application software that was hosted by a crashed computer will prevent the surviving part of the control system from fulfilling its role. Although after a crash the majority of the distributed computer control system's processing power is still available the control system will have been crippled by the loss of vital software components.

Whereas it is possible to design the executive software of a distributed computer control system so that the loss of one of its constituent kernels will not prevent the others from functioning normally, it is not possible to do this for the application software. Process survivability was conceived as a way of preventing application processes from being lost in a computer crash.

Process survivability is a way of making the application level of a distributed computer control system 'n out of m' crash tolerant: the computer control system is able to tolerate the crashing of n out of its m computers. Process survivability enhances a distributed computer control system's natural high availability by making it invulnerable to computer crashes.

11.2 A review of the work presented in this thesis

In this thesis we have established the need for process survivability in a distributed computer control system and we have identified a number of applications that could beneficially be based on such a control system. Previous work in this field has been described and we have related process survivability to this work.

We have described PROSUR the distributed computer control system that we specially designed in order to have an environment in which to design and develop process survivability.

The major part of this thesis is concerned with the implementation of process survivability in PROSUR. We have described how process-sets are implemented by the distributed kernel level, and how the process survivability level recovers the restarted application processes to a consistent state after a crash.

We believe, and we hope that it has been shown, that process survivability is practicable and that it could be implemented. However, the factor that would decide whether process survivability could be adopted or not, is the way in which the distributed computer control system's performance is affected by the overheads resultant from process survivability. A simulation study was performed to investigate this and the results of this study have been presented and some conclusions drawn.

11.3 An appraisal of process survivability

In Chapter 5 we specified that process survivability should ensure that all application processes survive multiple computer crashes (up to the limit of the processes' redundancy) no matter when these crashes occur, and that the provision of process survivability should be transparent to the application programmer. Furthermore, it was specified that each process

should have only a single outstanding recovery point and that processes should be recovered independently of each other, thereby avoiding the domino effect. We have been successful in meeting all of these requirements.

Another aim specified in Chapter 5 was to attempt to limit the overheads imposed on the control system as a result of supporting process survivability. Unfortunately, because of the presence of time-dependent functions in PROSUR's P/L it is necessary to perform a secure point prior to every SEND. This seems to be a very high rate, although it is the same as that found in Tandem. A way of reducing the secure point rate was however described, and others no doubt could be found.

The simulation study has not provided us with a categorical answer to the question of whether process survivability is practical or not, but then, it was not intended to. Instead it has provided us with a number of interesting indications as to process survivability's practicality over a range of different situations. Without repeating the summary of the previous chapter we would like to reiterate the major conclusion that we have drawn.

Process survivability is better suited to distributed computer control systems that have a high ratio of processes to computers. The performance obtained in these configurations is less affected by process survivability, and for high levels of redundancy the performance of these configurations is better than that of those configurations with a lower ratio of processes to computers. Even then, process survivability with high levels of redundancy may be precluded because of its effect on response time, which appears to increase linearly with increases in redundancy.

A further point to be made is that process survivability was not designed to cope with process death, and in order to prevent inconsistencies arising due to process death all processes have the same level of redundancy, and recovery is only guaranteed for as long as the number of computer crashes is within that level of redundancy. If process survivability were to be enhanced so that it could cope with process death, then each process could have its own level of redundancy based on its importance, and this should reduce the overheads. In Appendix H we briefly describe the inconsistencies that can arise after a process dies and we explain the potential set of circumstances that causes them to arise.

11.4 Future development

We believe that process survivability as it stands is a sound basis on which to develop a crash tolerant distributed computer control system. Rather than develop the combination of PROSUR and process survivability any further, we believe that it would be more profitable to add process survivability to an established distributed computer control system, perferably one with a definite application, as this would allow concentration on and direction of process survivability's future development.

Although process survivabiltiy was developed specifically for PROSUR this does not preclude its inclusion in other distributed computer control systems. Process survivability has been designed to be transparent to the application programmer. This transparency means that process survivability could be added to an existing distributed computer control system without having to rewrite application programs and re-train application programmers. Re-working would be limited to the distributed kernel level and to the hardware level so that they provide the support needed by the process survivability level. Although process survivability has been designed for a distributed programming language based on

asynchronous message passing we cannot see any reason why it could not be adapted to cope with any of the other interprocess communication mechanisms.

If process survivability is to be exploited to its full potential its future must be in this direction: it must be developed with a particular application in mind.

Appendix A. Class Construct

The class construct described below is based on those proposed by Wirth (1978) and Brinch Hansen (1977), and the terminology used to describe them is taken from Dahl and Hoare 1972.

A class type defines an abstract data type by grouping together a complex data structure and the procedures that manipulate that data structure. An instance of a class type is called an object.

The syntax of a class type is shown in Figure A.a. A class type consists of a class heading, input divisions, export divisions, declarations and main body. The variables and procedures declared within a class are its attributes. The import and export divisions define the interface between the class and the rest of the program.

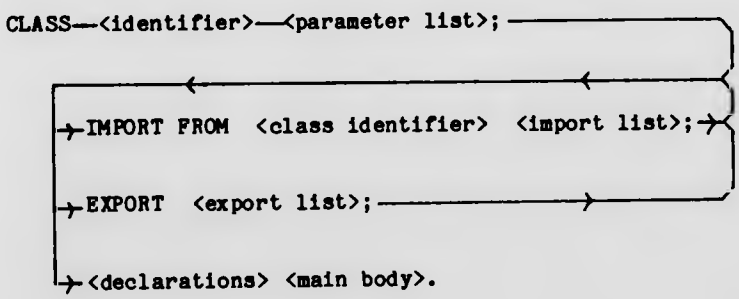


Figure A.a

The formal parameters in the class heading enable an object to be tailored to a specific task when it is created. The class's main body contains the code to initialise the object's data structure. To create an object the program executes the command:

INIT <objectname> (<actual parameters>)

where <objectname> has been declared to be an instance of some class type.

When INIT is called the object is created according to the parameter's specification and the class's main body is executed to initialise the object's data structure. Once an object has been created its data structure exists forever, and the data structure's values are maintained even when control is not within the class's code.

The Export division specifies those procedures declared within the class that can be used by the surrounding program and by other classes. Only those procedures that are explicitly exported by a class can be accessed. The other procedures and the data structure declared within a class are private.

The Import divisions specify those procedures that are used by the class and which are defined outside of it. These procedures can come from other classes and from the main program itself. Each source has a separate Import statement.

The main program implicitly imports all procedures exported by its classes. Any procedure declared globally in the program is implicitly exported to its classes and implicitly imported by its classes.

To use a procedure exported by an object the procedure's name is preceded by the name of the object (if any), for example:

```
To.Send ( .... )
```

would call the procedure 'Send' imported from an object called 'To'.

Appendix B. Exception Raising

In order to support exception raising each kernel maintains a number of tables and linked lists. Figure B.a illustrates these tables and lists for a particular kernel.

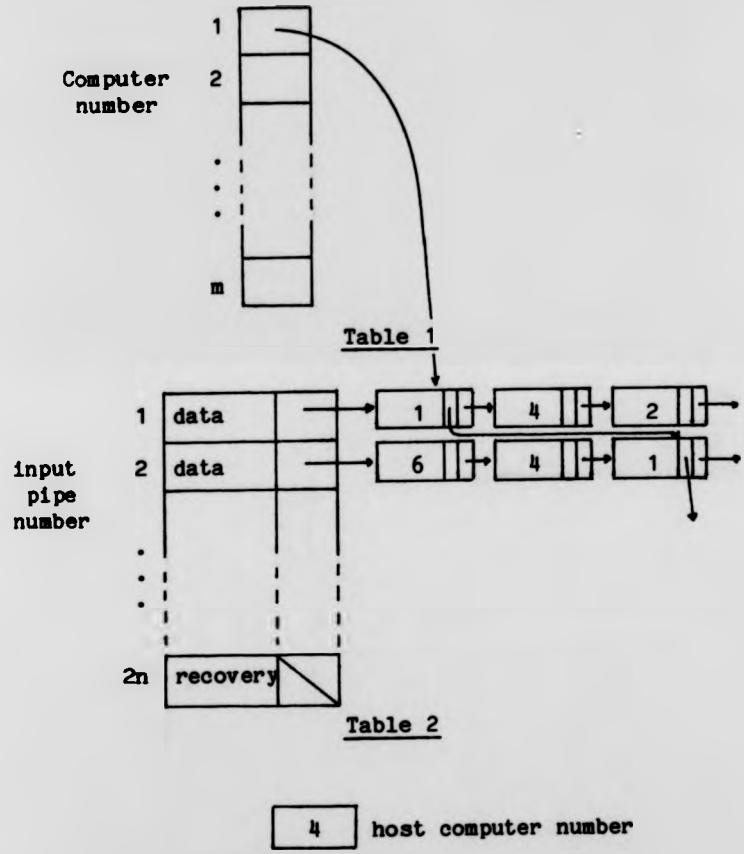


Figure B.a

The first of these tables, Table 2 in Figure B.a, is part of every primary replicate's and every backup replicate's process descriptor. This table defines for each of the replicate's input pipes whether that pipe is used to carry user messages (in which case it is a DataIn) or recovery messages (in which case it is a RcvyIn). The table also contains, for each

input pipe that carries user messages, a list of the computer addresses of the replicates of the sender process that is linked to that input pipe. These lists are maintained in the order in which the sender's replicates will be activated. There is no such list for input pipes carrying recovery messages.

Our example shows a particular replicate's table. The sender linked to this particular replicate by input pipe number 1 has replicates on computers 1, 4, 2 and others. The computer address of the sender's current primary replicate is the first number in the list - 1.

The second table, Table 1 in Figure B.a, links together all of the list elements for a particular computer; there is only one of these tables per computer. The final piece of information needed by a kernel is its host computer's address.

All of the information needed to fill these tables is provided by the system manager at system initialisation time.

Every time a kernel places a message into an input pipe it determines whether that input pipe carries recovery messages or user messages. If the former is true then that pipe must be a RcvyIn in an Ochannel class and so that class's exception is raised.

When the kernel detects the crash of a computer it uses its Table 1 to remove that computer's address elements from the input pipe's lists. If an element is removed and it is at the front of the list, and if the replicate to which the input pipe belongs is a primary replicate, then an exception is raised in the input channel class that contains that input pipe.

In our example Table 2 belongs to a primary replicate. If computer 1 crashes then the exception in the input channel class to which input pipe number 1 belongs will be raised. The kernel will pass to the exception handler the receiver's current computer address (4) and the new computer address of the sender - 4.

When a backup replicate is activated all of its input channel exceptions are raised because all of its input channels have broken. Each exception handler is passed the address of the backup replicate's host computer and the address of that channel's sender (the first entry in that class's input pipe's list). If in our example Table 2 were to belong to a backup replicate that is activated, then the values 4 and 1 would be passed to the exception handler of the input channel class that hosts input pipe 1.

In our example Table 2 belongs to a primary replicate. If computer 1 crashes then the exception in the input channel class to which input pipe number 1 belongs will be raised. The kernel will pass to the exception handler the receiver's current computer address (4) and the new computer address of the sender - 4.

When a backup replicate is activated all of its input channel exceptions are raised because all of its input channels have broken. Each exception handler is passed the address of the backup replicate's host computer and the address of that channel's sender (the first entry in that class's input pipe's list). If in our example Table 2 were to belong to a backup replicate that is activated, then the values 4 and 1 would be passed to the exception handler of the input channel class that hosts input pipe 1.

```
*************************
*                       *
*  The Input Channel Class  *
*                       *
*************************
```

```
Class Ichannel (size : Integer; basetype : Type);


Import From SP
  Timestamp, Execute;

Export
  RECEIVE, PENDING;

Const
  extsize        = size * 2;
  mRcount        = size;

Type
  extbasetype    = Record
                     seqno : Integer;
                     data  : basetype;
                   End;

  IPCstatus      = (succeeded, failed);

  RcvyMsge       = Record
                     seqno  : Integer;
                     loc    : Integer;
                   End;

Var
  DataIn         : kinputpipe [extsize] Of extbasetype;
  Rseqno         : Integer;
  Rcount         : Integer;
  RcvyOut        : koutputpipe Of RcvyMsge;
  timestamp      : Integer;
  Pflag          : Boolean;
  Pstore         : basetype;
```

```
{ RECEIVE and PENDING }

Procedure Read (Var message : basetype; Var status : IPCstatus;
                        timeout : Integer);

  Var
    m : extbasetype;

  Begin
      With Exceptionsoff Do
      Begin
        If timestamp < SP.Timestamp Then Begin
                                           Rcount := 0;
                                           kfree (DataIn);
                                           timestamp := SP.Timestamp;
                                         End;
        kread (DataIn, m, status, timeout);
        If status = succeeded Then Begin
                                     message := m.data;
                                     Rseqno := m.seqno;
                                     Rcount := Rcount + 1;
                                     If Rcount = mRcount Then SP.Execute;
                                   End;
      End;
  End;


Procedure RECEIVE (Var message : basetype; Var status : IPCstatus;
                         timeout : Integer);

  Begin
      If Pflag Then Begin
                      status  := succeeded;
                      message := Pstore;
                      Pflag   := false;
                    End
               Else Read (message, status, timeout);
   End;


Function PENDING : Boolean;

  Var
    status : IPCstatus;

  Begin
      If Pflag Then PENDING := true
                Else Begin
                       Read (Pstore, status, 0);
                       If status=succeeded Then Pflag := true
                                           Else Pflag := false;
                       PENDING := Pflag;
                     End;
  End;
```

```
                { The Exception Handler }

Exception Handler Recovery (rloc, sloc : Integer);

  Var
    m      : extbasetype;
    rm     : RcvyMsge;
    status : IPCstatus;

  Begin
    With Exceptionsoff Do
    Begin
      rm.loc := rloc;
      kpeeklast (DataIn, m, status, 0);
      If status = succeeded Then rm.seqno := m.seqno
                            Else rm.seqno := Rseqno;
      kconnect (RcvyOut, sloc);
      ksend (RcvyOut, rm, status, -1);
    End;
  End;



                { Main Body - initialisation code }

Begin
  Rseqno        := 0;
  Rcount        := 0;
  timestamp     := 0;
  Pflag         := false;
End.
```

```
*************************
*                       *
*  The Output Channel Class  *
*                       *
*************************
```

```
Class Ochannel (size : Integer; basetype : Type);


Import From SP
  Execute;

Export
  SEND;

Const
  extsize        = size * 2;

Insecure
  WaitRcvy       = true;

Type
  extbasetype    = Record
                     seqno : Integer;
                     data  : basetype;
                   End;

  IPCstatus      = (succeeded, failed);

  RcvyMsge       = Record
                     seqno  : Integer;
                     loc    : Integer;
                   End;

Var
  DataOut        : koutputpipe Of extbasetype;
  RcvyStore      : Record
                     fifo : Array [1..extsize] Of extbasetype;
                     top  : Integer;
                   End;
  Sseqno         : Integer;
  ngagged        : Integer;
  RcvyIn         : kinputpipe [MaxNoHits] Of RcvyMsge;
```

```
                { SEND }

Procedure SEND (message : basetype; Var status : IPCstatus;
                        timeout : Integer);

  Var
    m    : extbasetype;


  Procedure save (m : extbasetype);

    Begin
      { works with exception handling turned off by SEND }
      With RcvyStore Do
      Begin
        If top = extsize Then top := 1
                         Else top := top + 1;
        fifo[top] := m;
      End;
    End;

  Begin
    SP.Execute;
    Waituntil ( WaitRcvy = false );
    With Exceptionsoff Do
    Begin
      Sseqno := Sseqno + 1;
      m.seqno := Sseqno;
      m.data:= message;
      If ngagged > 0 Then Begin
                            status := succeeded;
                            ngagged := ngagged - 1;
                          End
                     Else ksend (DataOut, m, status, timeout);
      If status = succeeded Then save (m);
    End;
  End;
```

```
                { The Exception Handler }

Exception Handler Recovery;

  Var
    rm          : RcvyMsge;
    status      : IPCstatus;

    Procedure replace (sseq, rseq : Integer);

      Var
        i, j    : Integer;

      Begin
        { works with exception handling turned off by recovery }
        With RcvyStore Do
        Begin
          ngagged := 0;
          For i := (top + 1) - (sseq - rseq) To top Do
          Begin
            If i <= 0 Then j := extsize + i
                      Else j := i;
            If fifo[j].seqno > rseq
               Then ksend (DataOut, fifo[j], status, -1);
          End;
        End;
      End;


  Begin { Recovery }
    With Exceptionsoff Do
    Begin
      WaitRcvy := false;
      kreceive (RcvyIn, rm, status, 0);
      kconnect (DataOut, rm.loc);
      If Sseqno > rm.seqno Then replace (Sseqno, rm.seqno)
         Else If Sseqno < rm.seqno Then ngagged := rm.seqno - Sseqno
                 Else { do nothing, they are already consistent };
    End;
  End;  { Recovery }


                { The Main Body - initialisation code }


Begin
  Sseqno         := 0;
  RcvyStore.top  := 0;
  ngagged        := 0;
  WaitRcvy       := false;
End.
```

The Secure Pointing Class

```
Class SPoint ( );

Export
  Timestamp, Execute;

Var
  tstamp        : Integer;


Function Timestamp : Integer;

  Begin
    Timestamp := tstamp;
  End;


Procedure Execute;

  Begin
    tstamp := tstamp + 1;
    ksecurepoint;
  End;


Begin
  tstamp := 1;
End;
```

The Device Class

```
Class IOdevice (device : IOdevice; basetype, argtype : Type);


Export
  DOIO;

Type
  IOdevice      = (Lineprinter, disk, VDU ... );
  IOoperation   = (input, output, move, control);
  IOstatus      = (complete, failure ...       );


Procedure DOIO (operation : IOoperation; Var data : basetype;
                  Var status : IOstatus; Var arguments : argtype);

  Begin
    kdoio (operation, data, device, status, arguments);
  End;


Begin
  { empty }
End.
```

Appendix D. Performance Figures

All figures are shown to two decimal places.

Mean work period - 2 seconds
Secure point insertion - module-based

Mean

level of redundancy

configuration	0	1	2	3	4	6
4	25	25	25	25	na	na
8	49.32	49.40	49.19	49.18	49.45	49.30
16	90.28	93.80	96.08	89.30	94.00	90.68
32	176.81	178.85	162.36	158.72	146.44	114.44
192	954.02	262.32	140.35	92.78	71.38	48.25

Standard Deviation

level of redundancy

configuration	0	1	2	3	4	6
4	5.66	5.57	5.76	5.85	na	na
8	9.45	9.35	7.71	11.37	9.62	9.20
16	15.00	14.05	11.85	16.08	13.56	13.18
32	17.28	14.90	29.15	19.96	15.68	14.77
192	9.03	6.15	4.18	3.48	3.56	2.71

Mean work period - 2 seconds
Secure point insertion - worst-case

Mean

configuration	level of redundancy 0	1	2	3	4	6
4	25.00	25.00	24.98	25.00	na	na
8	49.32	49.33	49.47	49.05	49.68	48.66
16	90.28	94.74	93.29	90.70	83.83	62.81
32	176.81	172.61	142.52	107.46	87.31	62.39
192	954.02	262.32	140.35	92.78	71.38	48.25

Standard Deviation

configuration	level of redundancy 0	1	2	3	4	6
4	5.66	5.81	5.86	6.02	na	na
8	9.45	8.92	9.69	9.28	8.29	8.90
16	15.00	13.89	16.24	14.05	12.25	10.33
32	17.28	16.93	18.70	20.12	15.96	10.55
192	9.03	6.15	4.18	3.48	3.56	2.71

Mean work period - 0.5 seconds
Secure point insertion - module-based

Mean

configuration	level of redundancy 0	1	2	3	4	6
4	25.000	24.82	24.77	24.90	na	na
8	48.03	48.78	48.81	48.68	48.44	46.85
16	93.54	93.48	91.79	85.60	74.16	54.13
32	173.83	167.17	125.59	90.35	73.95	51.15
192	913.67	117.54	58.80	39.24	30.53	20.05

Standard Deviation

configuration	level of redundancy 0	1	2	3	4	6
4	3.11	3.46	3.58	3.14	na	na
8	6.16	5.00	5.40	4.36	5.18	4.38
16	7.22	7.38	7.47	7.92	6.63	6.34
32	11.08	9.48	8.47	6.11	6.00	5.65
192	4.35	1.50	1.12	0.79	0.78	0.46

Mean work period - 0.5 seconds
Secure point insertion - worst-case

Mean

configuration	level of redundancy 0	1	2	3	4	6
4	25.00	24.78	24.97	24.82	na	na
8	48.03	48.72	48.69	47.71	42.02	30.80
16	93.54	92.13	75.43	55.25	42.91	29.66
32	173.83	130.47	75.11	52.35	40.54	28.14
192	913.67	117.54	58.80	39.24	30.53	20.05

Standard Deviation

configuration	level of redundancy 0	1	2	3	4	6
4	3.11	3.23	3.16	3.33	na	na
8	6.16	5.18	4.87	3.86	4.76	4.09
16	7.22	7.64	6.93	5.11	5.06	4.20
32	11.08	7.50	6.83	4.76	4.46	3.21
192	4.35	1.50	1.12	0.79	0.78	0.46

Mean work period - 0.12 seconds
Secure point insertion - module-based

Mean

level of redundancy

configuration	0	1	2	3	4	6
4	24.59	24.74	24.75	24.61	na	na
8	48.89	48.28	47.97	43.78	36.45	25.74
16	95.00	90.79	65.28	46.53	35.52	24.12
32	177.68	113.73	61.71	42.95	32.90	22.01
192	551.88	50.17	24.84	16.46	12.78	8.54

Standard Deviation

level of redundancy

configuration	0	1	2	3	4	6
4	2.04	1.79	1.76	1.74	na	na
8	2.66	2.76	2.72	2.98	2.57	2.14
16	3.85	3.68	3.15	3.40	2.39	1.59
32	5.44	4.01	2.77	2.40	1.92	2.00
192	0.39	0.17	0.23	0.10	0.22	0.09

Mean work period - 0.12 seconds
Secure point insertion - worst-case

Mean

level of redundancy

configuration	0	1	2	3	4	6
4	24.59	24.71	24.63	23.12	na	na
8	48.89	48.18	38.84	27.26	20.86	14.09
16	95.00	70.20	37.64	25.62	19.41	13.24
32	177.68	67.13	34.22	23.45	17.99	12.15
192	551.88	50.17	24.84	16.46	12.78	8.54

Standard Deviation

level of redundancy

configuration	0	1	2	3	4	6
4	2.04	1.75	1.71	1.51	na	na
8	2.66	2.94	3.47	3.05	1.96	1.60
16	3.85	3.66	2.19	1.87	1.72	1.22
32	5.44	3.25	2.08	2.45	1.33	1.37
192	0.39	0.17	0.23	0.10	0.22	0.09

Appendix E. Accuracy of the Simulation Results

The performance of a configuration under a given set of circumstances is defined by a distribution X with a mean E[X]. We wish to estimate E[X] for each experiment performed.

Running an experiment once produces a sample from the distribution X. By repeating the experiment with different random number seeds we can produce a sample mean $\bar{x}$, which is the average of the samples taken.

The sample mean $\bar{x}$ is an unbiased estimator of E[X] (Sanders et al. 1976). We aimed to take sufficient samples from each distribution in order to achieve a 99% confidence level that E[X] will be within 0.5 seconds of the sample mean i.e.

$$P (\bar{x} - 0.5 < E[X] < \bar{x} + 0.5) = 0.99$$

Unfortunately, the amount of processing needed to achieve this level of accuracy for all 138 experiments was so great that the computer time and resources required to perform it were unavailable. Instead we first ran each experiment once so that the important ones could be identified, and then these 54 were repeated (each time with a different random number seed) to achieve the required accuracy. Even then it was not possible to do this for all 54, as some of them would have had to be run several hundred times.

Other less important experiments were run 11 times each, and others only once. In each case the sample mean is used to estimate E[X] although without any great accuracy.

The tables below show which experiments are accurate (those marked with a '*') and how many samples were taken from each of the others. To achieve this limited overall accuracy required just under 100 hours of CPU time on a CDC 7600.

Mean work period - 2 seconds
Secure point insertion - module-based

level of redundancy

configuration	0	1	2	3	4	6
4	1	11	11	11	na	na
8	1	11	1	1	11	*
16	1	11	1	1	11	11
32	1	11	1	1	11	11
192	1	11	1	11	11	11

Mean work period - 2 seconds
Secure point insertion - worst-case

level of redundancy

configuration	0	1	2	3	4	6
4	1	11	11	11	na	na
8	1	11	1	11	1	*
16	1	11	1	11	1	11
32	1	11	1	11	1	11
192	1	11	1	11	11	11

Mean work period - 0.5 seconds
Secure point insertion - module-based

level of redundancy

configuration	0	1	2	3	4	6
4	1	11	11	11	na	na
8	1	*	*	1	*	1
16	1	28	*	1	*	1
32	1	*	*	1	*	1
192	1	11	11	1	11	1

Mean work period - 0.5 seconds
Secure point insertion - worst-case

		level of redundancy					
		0	1	2	3	4	6
	4	1	11	11	11	na	na
	8	1	*	*	1	*	1
configuration	16	1	*	*	1	*	1
	32	1	31	*	1	*	1
	192	1	11	11	1	11	1

Mean work period - 0.12 seconds
Secure point insertion - module-based

		level of redundancy					
		0	1	2	3	4	6
	4	1	11	11	11	na	na
	8	1	*	*	1	*	1
configuration	16	1	*	*	1	*	1
	32	1	*	*	1	*	1
	192	1	11	11	1	11	1

Mean work period - 0.12 seconds
Secure point insertion - worst-case

		level of redundancy					
		0	1	2	3	4	6
	4	1	11	11	11	na	na
	8	1	*	*	1	*	1
configuration	16	1	*	*	1	*	1
	32	1	*	*	1	*	1
	192	1	11	11	1	11	1

Appendix F. Send, Receive and Secure Point Times

All figures are shown to two decimal places, or more where appropriate.

Mean work period - 0.5 seconds
Secure point insertion - worst-case

Mean Receive Time

configuration \ level of redundancy	0	1	2	3	4	6
4	9.07	9.81	9.48	9.23	na	na
8	6.00	5.52	5.20	4.84	5.46	7.09
16	3.07	2.83	3.19	4.27	5.43	7.70
32	1.63	1.85	3.11	4.28	5.54	7.43
192	0.09	0.84	1.36	2.08	2.38	3.50

Standard Deviation of Receive Time

configuration \ level of redundancy	0	1	2	3	4	6
4	16.71	18.79	18.39	17.28	na	na
8	11.31	10.21	9.94	9.12	10.33	13.86
16	5.59	5.34	6.07	7.92	10.34	15.14
32	2.91	3.42	6.01	8.41	10.58	13.76
192	0.14	1.43	2.44	3.62	4.07	6.12

Mean Send Delay

configuration	level of redundancy 0	1	2	3	4	6
4	0.000003	0.0057	0.013	0.026	na	na
8	0.000014	0.02	0.058	0.13	0.20	0.31
16	0.00006	0.084	0.21	0.27	0.32	0.36
32	0.00021	0.27	0.33	0.40	0.39	0.42
192	0.02	0.48	0.41	0.45	0.50	0.49

Standard Deviation of Send Delay

configuration	level of redundancy 0	1	2	3	4	6
4	0.00012	0.01	0.018	0.03	na	na
8	0.00029	0.028	0.065	0.14	0.21	0.30
16	0.00088	0.097	0.23	0.29	0.34	0.36
32	0.0023	0.31	0.35	0.40	0.38	0.43
192	0.05	0.70	0.67	0.72	0.79	0.78

Mean Secure Point Time

configuration	level of redundancy 1	2	3	4	6
4	1.98	3.92	5.42	na	na
8	1.00	2.21	3.87	5.99	10.43
16	0.85	2.84	5.11	7.27	11.38
32	1.41	3.85	6.20	8.31	12.93
192	3.22	7.87	12.13	16.22	25.14

Standard Deviation of Secure Point Time

configuration	level of redundancy 1	2	3	4	6
4	0.90	1.61	1.99	na	na
8	0.47	0.89	1.19	1.74	2.63
16	0.33	1.04	1.77	2.31	3.30
32	0.70	1.23	1.85	2.31	3.13
192	1.17	2.02	2.73	3.29	4.92

Appendix G. ALU Performance Figures

All figures are shown to two decimal places.

Mean work period - 0.5 seconds
Secure point insertion - worst-case

		level of redundancy					
		0	1	2	3	4	6
configuration	4	0.05	11.64	25.6	41.29	na	na
	8	0.09	23.68	53.41	77.99	88.19	94.05
	16	0.16	46.44	78.13	86.07	88.97	91.79
	32	0.28	55.63	74.15	80.56	82.73	85.77
	192	8.79	51.91	57.59	60.73	62.73	63.99

Appendix H. Process Death

If all of an application process's replicates are lost then that process is said to have died. When a process dies inconsistencies do not arise between the dead process and its communicants, but they can arise between two processes that communicate with each other via the dead process. There follows a description of how these inconsistencies arise.

Figure Ha shows three processes at the time of a crash. Prior to the crash Process-1 sent a message to Process-2, and then, as a direct result of receiving this message, Process-2 sent Process-3 a further message. The crash results in Process-1 being restarted, and in Process-2 dying. On restarting Process-1 will repeat its SEND and this will fail because Process-2 is dead. This leads to an inconsistency arising between Process-2 and Process-3 as Process-3 has received a message which with respect to Process-1's current state it should not have been sent.

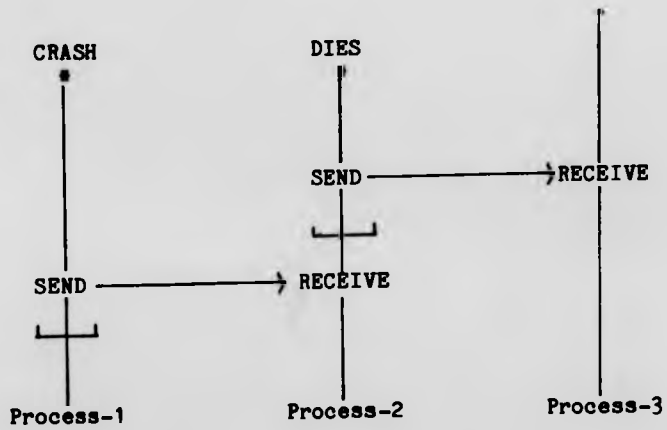


Figure H.a

This inconsistency cannot be prevented as the recovery of a process to a consistent state requires the assistance of that process's communicants. In our example Process-1 cannot be recovered by gagging its SEND, because Process-2 has died. The only way that this could be overcome is if a process's recovery were to be completely self-contained and not require assistance from its communicants.

References

The system of bibliographical citation used throughout the thesis is in the same style as that used by Lister in 'Fundamentals of operating systems' (Lister 1979). The abbreviations used in the journal titles are taken from 'The abbreviation of titles of periodicals' published by the British Standards Institution (1975).

Allworth, S.T. (1981), Introduction to real-time software design, The Macmillan Press Ltd., London

Alsberg, P.A. and Day, J.D. (1976), A principle for resilient sharing of distributed resources, Proc.2nd.Int.Conf. Software Engineering, San Francisco, 562-570

Anderson, T. and Kerr, R. (1976), Recovery blocks in action: a system supporting high reliability, Proc.2nd.Int.Conf. Software Engineering, San Francisco, 477-457

Anderson, T. and Lee, P.A. (1981), Fault tolerance: principles and practice, Prentice/Hall International, London

Anderson, T. and Lee, P.A. (1982), Fault tolerance terminology proposals, Dig.Pap. FTCS-12: 12th.Annu.Int.Symp. Fault-Tolerant Computing, Santa Monica (CA), 29-33

Anderson, T. and Moulding, M.R. (1982), Evaluating software fault tolerance in a real-time system, Internal Report, MARI, Newcastle upon Tyne

Aviziensis, A., Gilley, G.C., Mathur, F.P., Rennels, D.A., Rohr, J.A. and Rubin, D.K. (1971), The STAR (Self-Testing And Repairing) computer: an investigation of the theory and practice of fault-tolerant computer design, IEEE Trans. Computers, C-20(11), 1312-1321

Bartlett, J.F. (1978), A 'nonstop' operating system, Proc.11th. Hawaii Int.Conf. System Sciences, Honolulu, 103-117

Bartlett, J.F. (1981), A nonstop kernel, Proc.8th.Symp. Operating System Principles, Pacific Grove (CA), 22-29

Bennett, K.H. and Singleton, P. (1982), The design of a microprocessor-based access logic for the Cambridge Ring, J. Microcomputer Applications, 5, 195-207

Berthelsen, C. (1983), Disaster: it may never happen but..., Computer Systems, Dec., 29-32

Boebert, W.E., Franta, W.R., Jensen, E.D. and Kain, R.Y. (1978), Decentralized executive control in distributed computer systems, Proc. COMPSAC 78, Chicago, 254-258

Borg, A., Baumbach, J., and Glazer, S. (1983), A message system supporting fault tolerance, Proc.9th. ACM Symp. Operating System Principles, Bretton Woods (NH), 90-99

Bremen, C. (1983), Guarding against the day of disaster, Computing, 27 Oct.

Brenner, J.B., Burton, C.P., Kitto, A.D. and Portman, E.C.P. (1980), Project Little - an experimental ultra reliable system, ICL Technical J., 12(1), 47-58

Brinch Hansen, P. (1973), Operating system principles, Prentice-Hall Inc., Englewood Cliffs (NJ)

Brinch Hansen, P. (1977), The architecture of concurrent programs, Prentice-Hall Inc., Englewood Cliffs (NJ)

Brinch Hansen, P. (1978), Distributed Processes: a concurrent programming concept, Commun. ACM, 21(11), 934-941

British Standards Institution (1975), The abbreviation of titles of periodicals. Part 2. Word-abbreviation list, BS4148, pt.2, n.p.

Brownbridge, D.R., Marshall, L.F. and Randell, B. (1982), The Newcastle Connection or UNIXes of the world unite!, Software Practice and Experience, 12, 1147-1162

Burton, C.P. (1982), Private letter, Decision Support Unit, International Computers Ltd., Manchester, 30 Sept.

Carruthers, Cdr.J.F. (1979), Shinpads - a new ship integration concept, Naval Engineers J., April, 155-163

Carter, W.C. and Bouricius, W.G. (1971), A survey of fault-tolerant architecture and its evaluation, Computer, 4(1), 9-16

Carter, W.C., Putzolu, G.R., Wadia, A.B., Bouricius, W.G., Jessep, D.C., Hsieh, E.P. and Tan, C.J. (1977), Cost effectiveness of self checking computer design, Dig.Pap. FTCS-7: 7th.Annu.Int.Conf. Fault-Tolerant Computing, Los Angeles, 117-123

Casey, L. and Shelness, N. (1977), A domain structure for distributed computer systems, Proc.6th. ACM Symp. Operating System Principles, West Lafayette (IN), 101-108

Chen, L. and Aviziensis, A. (1978), N-Version programming: a fault-tolerance approach to reliability of software operation, Dig.Pap. FTCS-8: 8th.Annu.Int.Conf. Fault-Tolerant Computing, Toulouse, 3-9

Clark, D.D., Pogran, K.T. and Reed, D.P. (1978), An introduction to local area networks, Proc. IEEE, 66(11), 1487-1517

Cole, R. (1982), Computer communications, The Macmillan Press Ltd., London

Computing (1983), Racal develops local net for use by navy, Computing, 15 Sept.

Cristian, F. (1980), Exception handling and software-fault tolerance, Dig.Pap. FTCS-10: 10th.Int.Symp. Fault-Tolerant Computing, Kyoto, Japan, 97-103

Dahl, O.-J. and Hoare C.A.R. (1972), Hierarchical program structures, in Structured Programming (eds. O.-J.Dahl, E.W.Dijkstra and C.A.R.Hoare), Academic Press Inc. (London) Ltd., 175-220

Denning, P.J. (1968), The working set model for program behaviour, Commun. ACM, 11(5), 323-333

Farber, G. (1978), Principles and applications of decentralised process control computer systems, IFAC 1978, 1, Helsinki, 385-392

Feustel, E.A. (1973), On the advantages of tagged architecture, IEEE Trans. Computers, C-22(7), 644-656

Gaude, C., Girard, B., Langet, J., Palassin, S. and Kaiser, C. (1980), Design and appraisal of operating systems matched in selective active redundancy, Dig.Pap. FTCS-10: 10th.Int.Symp. Fault-Tolerant Computing, Kyoto, Japan, 78-80

Gee, K.C.E. (1983), Introduction to local area computer networks, The Macmillan Press Ltd., London

Goodenough, J.B. (1975), Exception handling: issues and a proposed notation, Commun. ACM, 18(12), 683-696

Gray, J.N. (1978), Notes on database operating systems, in Operating systems: an advanced course (eds. R.Bayer, R.M.Graham and G.Seegmuller), Springer-Verlag, Berlin, 393-481

Harland, D.M. (1981), On facilities for handling exceptions and preventing deadlock in a system of concurrent processes, Internal Report, Department of Computational Science, University of St.Andrews, Scotland

Harper, M.E. (1982), Mutual exclusion within both software - and hardware - driven kernel primitives, ACM Operating Systems Review, 16(4), 60-68

Harrington, T. (1983), Gauging the odds in the back-up business, Computing, 24 Mar., 30-31

Hill, J.S. and Stainsby, M.G. (1980), ASWE Serial Highway: a highway for intercomputer communication, J. Naval Science, 6(3), 216-221

Hoare, C.A.R. (1978), Communicating sequential processes, Commun. ACM, 21(8), 666-677

Holler, E. (1983), Multiple copy update, in Distributed systems - architecture and implementation (eds. B.W.Lampson, M.Paul and H.J.Siegert), Springer-Verlag, Berlin, 284-307

Hopkins, A.L. and Smith, T.B. (1975), The architectural elements of a symmetric fault-tolerant multiprocessor, IEEE Trans. Computers, C-24(4), 498-505

Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B. (1974), A program structure for error detection and recovery, Proc.Conf. Operating Systems: Theoretical and Practical Aspects, Rocenquourt, France, 177-193

Jensen, E.D. (1978), The Honeywell Experimental Distributed Processor - an overview, Computer, 11(1), 28-38

Jensen, E.D. (1983), Distributed control, in Distributed systems - architecture and implementation (eds. B.W.Lampson, M.Paul and H.J.Siegert), Springer-Verlag, Berlin, 175-190

Jensen, E.D., Anderson, A. and Marshall, G.D. (1977), Feasibility demonstration of distributed processing for small ship command and control, IEEE Computer Science Repository., Rep.R77-121

Jensen, K. and Wirth, N. (1978), Pascal user manual and report, Springer-Verlag, New York

Johnson, M.A. (1980) Ring byte stream protocol specification, Internal Report, Computer Laboratory, University of Cambridge

Jones, A.K. (1978), The object model: a conceptual tool for structuring software, in Operating systems: an advanced course (eds. R.Bayer, R.M.Graham and G.Seegmuller), Springer-Verlag, Berlin, 7-16

Kant, K. (1983), Efficient local checkpointing for software fault tolerance, ACM Operating Systems Review, 17(2), 11-13

Katzman, J.A. (1978), A fault-tolerant computing system, Proc.11th. Hawaii Int.Conf. System Sciences, Honolulu, 85-102

Kernighan, B.W. and Ritchie, D.M. (1978), The C programming language, Prentice-Hall Inc., Englewood Cliffs (NJ)

Kohler, W.H. (1981), A survey of techniques for synchronization and recovery in decentralised computer systems, Computing Surveys, 13(2), 149-183

Kramer, J., Magee, J. and Sloman, M. (1981), Intertask communication primitives for distributed computer control systems, 2nd.Int.Conf. Distributed Computing Systems, Paris

Kramer, J., Magee, J., Sloman, M. and Lister, A. (1982), CONIC: an integrated approach to distributed computer control systems, Internal Report, Department of Computing, Imperial College of Science and Technology, London

Lakin, W. (1982), Private Conversation, Director, ADNET Project, ASWE, Portsmouth, 26 Nov.

Lauer, H.C. and Needham, R.M. (1979), On the duality of operating system structures, Proc.2nd.Int.Symp. Operating Systems, reprinted in ACM Operating Systems Review, 13(2), 3-19

Lawrence, A. (1983), There's a lesson to be learnt from every disaster, Datalink, 2 May, 10

Le Lann, G. (1979), An analysis of different approaches to distributed computing, Proc.1st.Int.Conf. Distributed Processing Systems, Huntsville (ALA), 222-232

Le Lann, G. (1983), Synchronization, in Distributed systems - architecture and implementation (eds. B.W.Lampson, M.Paul and H.J.Siegert), Springer-Verlag, Berlin, 266-283

Lee, P.A., Ghani, N. and Heron, K. (1980), A recovery cache for the PDP-11, IEEE Trans. Computers, C-29(6), 546-549

Licklider, J.C.R. and Vezza, A. (1978), Applications of information networks, Proc. IEEE, 66(11), 1330-1346

Liskov, B. (1979), Primitives for distributed computing, Proc.7th.Symp. Operating System Principles, Pacific Grove (CA) ,33-42

Lister, A.M. (1979), Fundamentals of operating systems, 2nd.ed., The Macmillan Press Ltd., London

Lister, A., Magee, J., Sloman, M. and Kramer, J. (1980), Distributed process control systems: programming and configuration, Internal Report, Department of Computing and Control, Imperial College, London

Liu, M.T. and Reames, C.C. (1977), Message communication protocol and operating system design for the Distributed Loop Computer Network (DLCN), Proc.4th.Annu.Symp. Computer Architecture, n.p., 193-200

Mackie, D. (1978), The Tandem 16 NonStop system, State of the art report on system reliability and integrity, 2, Infotech, Maidenhead, 163-279

MacLaren, M.D. (1977), Exception handling in PL/1, SIGPLAN Notices, 12(3), 101-104

Mathur, F.P. and Aviziensis, A. (1970), Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair, AFIPS Spring Joint Computing Conf.Proc., 36, 375-383

McDermid, J.A. (1980), Checkpointing and error recovery in distributed systems, Internal report, Royal Signals and Radar Establishment, Malvern

McDermid, J.A. (1981a), Checkpointing and error recovery in distributed systems, Proc.2nd.Int.Conf. Distributed Computing Systems, Paris, 271-282

McDermid, J.A. (1981b), Error recovery techniques for fault tolerant distributed computer systems, Ph.D. Thesis, University of Birmingham

Melliar-Smith, M.P. and Randell, B. (1977), Software reliability: the role of programmed exception handling, SIGPLAN Notices, 12(3), 95-100

Menasce, D.A. (1978), Coordination in distributed systems: concurrency, crash recovery and database synchronization, Ph.D. Thesis, Univ. California at Los Angeles

Meraud, C., Browaeys, F. and Germain, G. (1976), Automatic rollback techniques of the COPRA computer, Proc. FTCS-6: 1976 Int.Symp. Fault-Tolerant Computing, Pittsburgh (PA), 23-29

Meraud, C. and Lloret, P. (1978), COPRA: a modular family of reconfigurable computers, Proc. IEEE 1978 National Aerospace and Electronics Conf., Dayton (OH), 822-827

Meraud, C., Browaeys, F., Queille, J.P. and Germain, G. (1979), Hardware and software design of the fault tolerant computer COPRA, Dig.Pap. FTCS-9: 9th.Ann.Int.Symp. Fault-Tolerant Computing, Madison (WI), 167

Merlin, P.M. and Randell, B. (1978), Consistent state restoration in distributed systems, Dig.Pap. FTCS-8: 8th.Ann.Int.Conf. Fault-Tolerant Computing, Toulouse, 129-134

Metcalfe, R.M. and Boggs, D.R. (1976), Ethernet: distributed packet switching for local computer networks, Commun. ACM, 19(7), 395-404

Miles, J. (1980), ASWE Mascot operating system: reference manual, Internal report, ASWE, Portsmouth

Ministry of Defence (1981), The ASWE Serial Highway, Defence Standard 00-19/Issue, MoD, Directorate of Standardisation, London

Moulding, M.R. (1980a), Reliability in multi-computer systems, Internal report, ASWE, Portsmouth

Moulding, M.R. (1980b), The ADNET communications system, Internal report, ASWE, Portsmouth

O'Brien, F.J. (1976), Rollback point insertion strategies, Proc. FTCS-6: 1976 Int.Symp. Fault-Tolerant Computing, Pittsburgh (PA), 138-142

Paker, Y. (1983), Multimicroprocessor systems, Academic Press, London

Plugge, W.R. and Perry, M.N. (1961), American Airlines' "SABRE" electronic reservations system, Proc. Western Joint Computer Conf., Los Angeles, 593-602

Popek, G.J. and Kline, C.S. (1978), Issues in kernel design, in Operating systems: an advanced course (eds. R.Bayer, R.M.Graham and G.Seegmuller), Springer-Verlag, Berlin, 209-227

Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G. (1981), LOCUS: a network transparent, high reliability distributed system, Proc.8th.Symp. Operating System Principles, Pacific Grove (CA), 169-177

Powell, M.L. and Presotto D.L. (1983), Publishing: a reliable broadcast communication mechanism, Proc.9th. ACM Symp. Operating System Principles, Bretton Woods (NH), 100-109

Prince, S. and Sloman, M. (1981), The communication requirements of a distributed computer control system, IEE Proc., 128, pt.E(1), 21-34

Randell, B. (1975), System structure for software fault tolerance, IEEE Trans. Software Engineering, SE-1(2), 220-232

Randell, B., Lee, P.A. and Treleaven, P.C. (1978), Reliability issues in computing system design, Computing Surveys, 10(2), 123-165

Rashid, R.F. (1980), A network operating system kernel for SPICE/DSN, Internal report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh (PA)

Rennels, D.A. (1978), Architectures for fault-tolerant spacecraft computers, Proc. IEEE, 66(10), 1255-1268

Richards, M. and Whitby-Strevens, C. (1980), BCPL the language and its compiler, Cambridge University Press, Cambridge

Rohr, J.A. (1973), STAREX self-repair routines: software recovery in the JPL-STAR computer, Digest of Papers FTC/3: 73 Int.Symp. Fault-Tolerant Computing, Palo Alto (CA), 11-16

Rowe, L.A., Hopwood, M.D. and Farber, D.J. (1973), Software methods for achieving fail-soft behaviour in the distributed computing system, Proc. IEEE Symp. Computer Software Reliability, n.p., 7-11

Rowland, T. (1982), Falklands disasters highlight the need for a review of warship's self defence, Electronic Times, 17 June, 8-9

Rudisin, G.J. (1980), Architectural issues in a reliable distributed file system, M.Sc. Thesis, Computer Science Department, Univ. California at Los Angeles

Russell, D.L. (1977), Process backup in producer-consumer systems, Proc.6th.Symp. Operating System Principles, West Lafayette (IN), 151-157

Russell, D.L. (1980), State restoration in systems of communicating processes, IEEE Trans. Software Engineering, SE-6(2), 183-194

Russell, D.L. and Tiedeman, M.J. (1979), Multiprocess recovery using conversations, Dig.Pap. FTCS-9: 9th.Ann.Int.Symp. Fault-Tolerant Computing, Madison (WI), 106-109

Saltzer, J.H. (1978), Research problems of decentralized systems with largely autonomous nodes, ACM Operating Systems Review, 12(1), 43-52

Sanders D.H., Murph A.F. and Eng R.J. (1976), Statistics: a fresh approach, McGraw-Hill Book Co., New York

Shanker, K.S. (1977), The total computer security problem: an overview, Computer, 10(6), 50-73

Shrivastava, S.K. (1978), Sequential Pascal with recovery blocks, Software Practice and Experience, 8, 177-185

Sincoskie, W.D. and Farber, D.J. (1980), SODS/OS: a distributed operating system for the IBM Series/1, ACM Operating Systems Review, 14(3), 46-54

Sloman, M.S. (1982), The CONIC communication system for distributed process control, Internal report, Imperial College, London

Solomon, M.H. and Finkel, R.A. (1979), The Roscoe distributed operating system, Proc.7th.Symp. Operating System Principles, Pacific Grove (CA), 108-114

Spector, A.Z. and Schwarz, P.M. (1983), Transactions: a construct for reliable distributed computing, ACM Operating Systems Review, 17(2), 18-35

Spirn, J.R. (1977), Program behaviour: models and measurements, Elsevier, North-Holland Inc., New York

Staunstrup, J. (1982), Message passing communication vs procedure call communication, Software Practise and Experience, 12, 223-234 223-234

Stepczyk, F. (1978), A case study in real-time distributed processing design, Proc. COMPSAC 78, Chicago, 514-519

Stroustrup, B. (1982), An experiment with the interchangeability of processes and monitors, Software Practice and Experience, 12, 1011-1025

Tillman, P.R. (1982), ADDAM - the ASWE distributed database management system, in Distributed databases (ed. H.J.Scheider), North-Holland Pub.Co., Amsterdam, 185-196

Tillman, P.R., Lakin, W.L., Sampson, K.F., Miles, J.A., Anderson, A. and Adcock, A. (1981), Software technology for naval applications: an interim report on ADNET, Internal report, ASWE:Software Sciences Ltd., Portsmouth

Von Linde, O.B. (1979), Computers can now perform vital functions safely, Railway Gazette International, 135(11), 1004-1006

Watson, R.W. (1983), Distributed system architecture model, in Distributed systems - architecture and implementation (eds. B.W.Lampson, M.Paul and H.J.Siegert), Springer-Verlag, Berlin, 10-56

Wegner, P. (1980), Programming with Ada: an introduction by means of graduated examples, Prentice-Hall Inc., Englewood Cliffs (NJ)

Wilkes, M.V. and Wheeler, D.J. (1979), The Cambridge digital communication ring, Proc. Local Area Network Symp., Boston, U.S.Bureau of Standards

Wirth, N. (1978), Modula-2, Internal report, Institut fur Informatik, Eidgenossische Technische Hockshule, Zurich

Wong, K. (1983), When bombs, fire and flood hit the system, Computer Weekly, 10 Nov., 44-46