Replication and Fault-Tolerance in Real-Time Systems.

by

Adrian Waterworth

Ph.D Thesis

September 1992

University of Newcastle upon Tyne Department of

Computing Science

Abstract.

The increased availability of sophisticated computer hardware and the corresponding decrease in its cost has led to a widespread growth in the use of computer systems for real-time plant and process control applications. Such applications typically place very high demands upon computer control systems and the development of appropriate control software for these application areas can present a number of problems not normally encountered in other applications.

First of all, real-time applications must be correct in the time domain as well as the value domain: returning results which are not only correct but also delivered on time. Further, since the potential for catastrophic failures can be high in a process or plant control environment, many real-time applications also have to meet high reliability requirements. These requirements will typically be met by means of a combination of fault avoidance and fault tolerance techniques.

This thesis is intended to address some of the problems encountered in the provision of fault tolerance in real-time applications programs. Specifically, it considers the use of replication to ensure the availability of services in real-time systems. In a real-time environment, providing support for replicated services can introduce a number of problems. In particular, the scope for non-deterministic behaviour in real-time applications can be quite large and this can lead to difficulties in maintaining consistent internal states across the members of a replica group. To tackle this problem, a model is proposed for fault tolerant real-time objects which not only allows such objects to perform application specific recovery operations and real-time processing activities such as event handling, but which also allows objects to be replicated. The architectural support required for such replicated objects is also discussed and, to conclude, the run-time overheads associated with the use of such replicated services are considered.

Acknowledgements.

First of all, I would like to thank my supervisor, Professor Santosh Shrivastava, who suggested this area of research and with whom I have had countless discussions over the years. His guidance and advice have been invaluable, as have his comments upon the numerous drafts of this thesis. Further thanks go to Dr. Paul Ezhilchelvan, who has also given of his time to read and comment upon the work presented here.

I am also grateful to all of my colleagues in the Computing Laboratory for providing me with such a stimulating and friendly working environment. Although there are far too many people to mention all of them individually, special thanks must go to Dr. Mark Little, Dr. Stuart Wheater and Dr. Graham Parrington, for numerous technical discussions over the years, and Peter Barrett of the Dependable Computing Systems Centre, for helping to clarify my understanding of the Delta-4 system.

Finally, I am deeply indebted to my family and my friends for all of the support and encouragement they have given me during my studies. In particular, I would like to thank my parents, who have helped me through all of the trials and tribulations of the past four years and without whom none of this would have been possible.

Financial support for much of the work described in this thesis was provided by a grant from the Science and Engineering Research Council, with further thanks being due to the British Aerospace Dependable Computing Systems Centre for my employment over the past year.

Table of Contents.

List of Figures.

Chapter 1.

Introduction.

The increased availability of sophisticated computer hardware and the corresponding decrease in its cost has led to a widespread growth in the use of computer systems for real-time plant and process control applications. For example, modern manufacturing cells make extensive use of industrial robots with microprocessor-based controllers, while flight control systems in military and civilian aircraft have grown in sophistication from simple trim controllers to complex auto-pilots and automatic landing systems. Such applications typically place very high demands upon computer control systems and the development of appropriate control software for these applications can present a number of problems not normally encountered in other application areas. [Stankovic 88a]

First of all, real-time programs are expected to perform correctly in the time domain as well as the value domain. That is to say, it is not sufficient for a particular operation simply to return the correct value when it is carried out, it must also return that value within some specified time interval. In fact, there may be cases where returning an approximate, or even inaccurate, value on time is preferable to returning an accurate value too late. Secondly, real-time applications may also have higher reliability requirements than "ordinary", non-real-time applications. For instance, flight control, nuclear plant control and weapons systems are just three examples of applications where the consequences of a failure may be disastrous and where the reliability of both the system hardware and control software must be ensured. Finally, the development of real-time software is complicated by the nature of the real-time environment itself. Hardware in a large real-time system may be distributed, consisting of several computers and a number of intelligent device controllers all connected by some kind of network. Control software must be appropriate to the underlying hardware configuration and it must be able to process incoming sensor signals from the environment and provide appropriate driver outputs for actuators.

This thesis considers some of the difficulties associated with the implementation of dependable real-time control software. In particular, it considers the use of replication to provide fault tolerance in real-time applications programs. In order to tolerate component failures, some form of replication scheme must be used, however supporting replicated components in a real-time environment can present a number of problems. This thesis examines the problems that can arise and proposes some possible solutions. The research areas covered include the development of an appropriate model for real-time objects and an analysis of the overheads associated with replicated services based upon that object model. First however, the remainder of this chapter gives a brief introduction to some of the underlying concepts of real-time computing and dependability.

1.1. Real-Time Systems.

There are many different definitions of the term "real-time system" however, for the purposes of this thesis, the following definitions, based upon those of [PDCS 90], will be adopted. A *real-time service* can be defined as a service that is required to be delivered within time intervals dictated by its environment[1]. A *real-time system* is then any system that delivers at least one real-time service. In other words, a real-time system can be loosely defined as a system that must interact with an external environment within defined timing constraints.

In general, it is possible to identify two distinct classes of real-time system. A *hard* real-time system is one in which at least one of its timing failure modes is costly or damaging to the system's environment. Hard systems therefore need to be able to offer guaranteed performance and design-time assurances of both timeliness and correctness should, ideally, be obtained. A *soft* real-time system, on the other hand, is one in which all possible timing failures are benign (i.e. delivery of a service at the wrong time may be useless but it will not be catastrophic). Hence, a soft system will not need to offer the same level of guaranteed performance as a hard system, however the basic design problem remains the same. The

[1]Note that to deliver a service may mean to sample an input rather than to produce an output.

system should be able to respond appropriately to events in its environment and produce outputs that are not only correct in value, but also delivered at the correct time.

Scheduling a set of real-time services in such a way that they all meet their timing constraints can be a difficult problem and, in some cases, a feasible schedule may not exist at all. However, a number of effective real-time scheduling mechanisms are already known and research in this particular field still continues. Although a full description of the current state of the art in real-time scheduling theory would be beyond the scope of this thesis, some of the fundamental concepts of real-time computing and real-time scheduling are discussed in more detail at the beginning of the following chapter.

1.2. Objects and Actions in Real-Time Programs.

At the heart of any application or system program can be found some form of abstract computational model or program structuring technique upon which the detailed implementation of that software is based. One such computational model that has grown in popularity in recent years is the *object-based* or *object-oriented* model, in which programs are structured as collections of interacting or communicating objects. An *object* consists of an internal state and a set of operations (or *methods*) that characterise its externally visible behaviour. Objects can only interact by means of invoking one another's methods and the internal implementation details of an object are hidden from the users of that object. Hence, objects provide the two highly desirable properties of *encapsulation* and *data abstraction.* Also, since all communication between objects takes place through method invocations, the flow of information in an object-based system is confined to specific, well-defined pathways. This not only prevents unwanted interactions between different parts of the system, it also makes it easier to provide transparent support for distributed applications since remote invocations can be identified and an appropriate remote procedure call [Birrell 84] [Shrivastava 82] or network message passing mechanism used.

1.2.1. Objects in Real-Time Systems.

From the point of view of real-time applications, adopting an object-based computational model and structuring programs in terms of interacting objects can offer a number of advantages. First of all, by having objects correspond to specific device or sub-system controllers, the logical structure of the control software can be made to reflect the physical structure of the system hardware. The control laws and interactions between different parts of the system can then be represented in the method invocations between different objects. Secondly, by allowing objects to have internal timing properties and including implicit or explicit scheduling parameters in method invocations, timing constraints on real-time services can be expressed as an integral part of the application software. Appropriate scheduling mechanisms can then be employed to ensure that the specified constraints are met, if possible. Finally, if objects are allowed to be *active*, that is if they are allowed to contain internal threads of control that carry out processing independently of incoming method invocations, then continuous monitoring or cyclic control law processing can be carried out within the object-based framework. This allows such activities to be assigned to appropriate objects, thereby maintaining the correspondence between the logical structure of the control software and the overall structure of the whole system.

1.2.2. Using Transactions.

Another advantage of the object-based approach is that the interfaces between objects provide a natural boundary for the containment of faults or errors. This can be exploited by structuring the activity of an application in terms of *transactions* or *atomic actions* (see, for example, [Wheater 90]). A transaction is, essentially, an all-or-nothing operation that either succeeds, having its intended effect, or fails, having no effect at all. This behaviour is usually achieved by means of a checkpoint and recovery mechanism that takes a copy of local state information before an operation begins and restores the system to that state if the operation fails. If all method invocations are executed as transactions, both error recovery within objects and error confinement across object boundaries can be supported. Hence, transactions can be used to ensure that objects only undergo correct and consistent state changes.

1.3. Exceptions and Replication.

Although transactions are one good way to support fault tolerance in programs, there are other techniques which can also be used, either to complement the transaction mechanism or to provide fault tolerance coverage for situations that the transaction mechanism cannot handle. For example, exception handling can be used to provide a flexible mechanism for error detection and recovery [Cristian 89][Campbell 86], while the availability of software components (objects, processes) can be ensured by replicating them on distinct processing nodes in a distributed or multiprocessor system (see, for example, [Little 90][Little 92]). Both of these techniques have been used in a number of systems and they are of particular interest for the real-time applications domain.

1.3.1. Exception Handling.

Like transactions, exception handling can be viewed as a structuring method for fault tolerant programs. The basis of the exception handling technique is the identification of *anticipated exceptional inputs* for which services cannot return correct results. For example, consider a procedure that takes two real numbers, x and y, and divides one by the other to return x÷y. An anticipated exceptional input for this "service" would be y=0, which under normal circumstances would lead to a system error. However, by defining a "Divide-By-Zero" exception that is raised by the division procedure whenever it is passed a divisor of zero, the error can be trapped and signalled back to the caller. The caller can then include an exception handler that implements an appropriate recovery strategy for this particular error.

The major advantage of exception handling is that it allows a high degree of flexibility in error recovery, since both exceptions and handlers can be freely defined by the application programmer and used wherever they are needed. However, there is also the disadvantage that exceptions, by definition, can only be defined for anticipated faults or errors.

Fortunately, this particular problem can be overcome by combining exception handling with some other fault tolerance technique, such as transactions, that can handle unanticipated errors. In fact, the combination of exceptions and transactions can be particularly effective, giving a unified fault tolerance mechanism that is both flexible and widely applicable.

1.3.2 Replication.

While techniques such as transactions and exception handling can help to ensure that an object provides correct service when requested, they cannot ensure the continued availability of an object when it is needed. In a distributed system, it is possible for some processing nodes to fail while others remain in service. When this happens, any objects that were resident on a failed node will, necessarily, be unavailable for as long as the node is off-line and requests intended for those objects will either be lost or remain unanswered until the node is repaired. If a system is intended to meet high reliability requirements, this problem must be addressed and an appropriate mechanism must be used to ensure object availability. The only suitable technique for this task is the use of some form of replication scheme, whereby several copies of an object are maintained within the system in order to ensure that the loss of a single copy (or some bounded number of copies) will not lead to a loss of the services which the object provides.

Unfortunately, replicating objects for availability is not as simple as it may, at first, seem. First, a multicast communications protocol may be needed to handle communication between groups of replicas or between single objects and groups. Second, replica group management protocols are required to ensure that the internal states of all the members of a group remain mutually consistent. This not only includes state information regarding incoming invocations and replies from other groups, but also changes in the membership of the group itself due to node failures and subsequent repairs. Finally, a number of problems can arise with non-determinism in replica groups. If two replicas can observe events in a different order, for example by receiving invocations in a different order due to differing communication delays, then the internal states of those two replicas can diverge and the replicas will produce different outputs. This is, of course, unacceptable, but the potential for such state divergence can be very high. Incoming messages, whether invocations or replies, event signals from the system's environment, interactions with a local clock or timebase and non-deterministic language constructs are all potential sources of divergence. Hence, to be effective, replica management protocols must also include mechanisms to handle problems such as these. The practical details of object replication therefore turn out to be far more complex than the basic concept.

1.4. Thesis Aims.

As described earlier, this thesis attempts to address some of the problems associated with the provision of fault tolerance in real-time applications software. Specifically, it considers the case where an application may be running on a distributed system, allowing replication techniques to be used to enhance the availability of software components (objects). The major research areas covered include :

- Development of an object model which is appropriate for use in a real-time environment and which not only allows objects to engage in real-time processing activities but also allows them to be replicated for availability.
- Provision of an integrated atomic action and exception handling mechanism within the framework of the real-time object model
- An examination of the architectural support required for implementing such objects, as well as the overheads associated with replicated real-time services.

1.5. Thesis Structure.

Chapter 2 begins with a more extensive examination of some of the basic concepts of real-time computing, before moving on to a more detailed discussion of atomic actions and exception handling mechanisms. Brief descriptions of three existing fault tolerant real-time operating systems are also given.

Issues relating to object replication are discussed in chapter 3. Passive and active replication strategies are described and their relative suitability for use in real-time applications assessed. The problems arising from replica state divergence are examined in more detail and appropriate methods for dealing with such state divergence are described.

Following on from this, chapter 4 introduces the real-time object model and programming notation which has been developed. Small examples demonstrate the usefulness of the features which have been incorporated into the model, as well as showing their use at the language level. Chapter 5 builds upon this by presenting some more complete, larger examples of real-time application programs.

In chapter 6, the implementation of the proposed object model will be considered. This will include an examination of the way in which the mechanisms that have been included in the model could be implemented, along with a discussion of the underlying architectural support that would be required if objects are to be replicated for availability. Chapter 7 then continues on the subject of replication, considering the overheads associated with the use of replicated, as opposed to non-replicated, services.

Finally, chapter 8 gives a summary of the thesis, including the main contributions of this research thus far, along with an outline of further work which remains to be done in the areas covered by this thesis.

Chapter 2.

Real-Time Systems and Fault-Tolerant Programs.

The previous chapter gave a brief definition of the term "real-time system" and introduced the concepts of atomic actions and exception handling as methods for supporting fault tolerance in programs. This chapter expands upon those basic ideas, considering real-time services and real-time scheduling in a little more detail before moving on to concentrate on fault tolerance issues and discussing both atomic actions and exception handling at greater length. The chapter concludes with brief descriptions of three existing fault tolerant real-time systems.

2.1. Real-Time Services.

As defined earlier, a real-time service is one that is required to be delivered within time intervals dictated by its environment. However, this definition can cover a wide range of different service requirements. It will therefore be worthwhile to take a closer look at the overall concept of real-time services, beginning with an examination of their defining characteristic: timing constraints.

2.1.1. Specifying Timing Constraints.

The timing constraints on a real-time task are usually specified by giving a *liveline*, which defines the earliest time at which the service may be delivered, and a *deadline*, which defines the latest time at which it may be delivered. It is also possible to specify a *targetline*, which is the time at which the system designer actually intends the service to be delivered. Such constraints can be represented using a graphical notation, due to Jensen et al [Jensen 85], in which the value or utility of the service is considered as a function of time. For example, figure 2.1 represents a real-time service that has a liveline of t_1 and a deadline of t_2. The service is only useful if it is delivered within the interval $[t_1,t_2]$ and the targetline for the service should, by definition, lie within this interval. Note that the graph has been extended so that it also shows the cost of never delivering the service. This is to allow for certain classes of real-time service where the cost of not delivering the service at all differs from the cost of delivering the service at the wrong time.

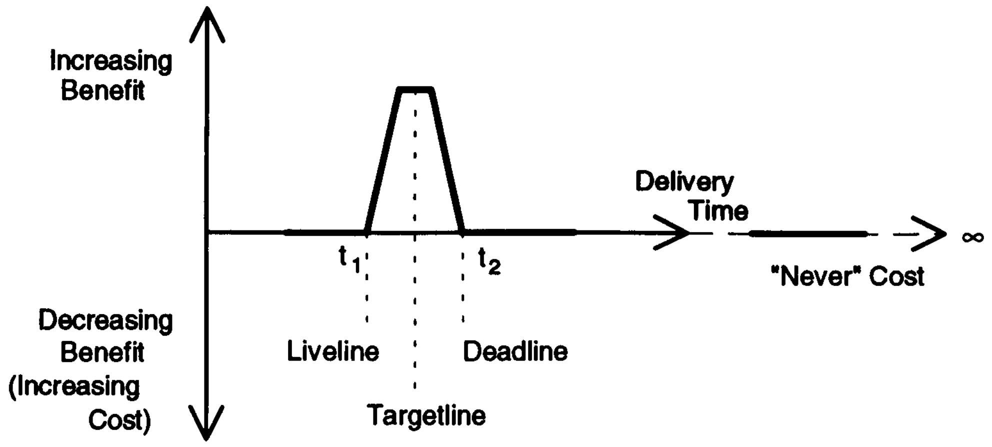


Figure 2.1. Value of real-time service vs. delivery time.

2.1.2. Periodic and Aperiodic Services.

Although livelines and deadlines can be used to define bounds on the delivery of a real-time service, they do not capture the long term behaviour of the service or express its more general timing properties. In particular, they do not capture the notion of periodicity. Real-time services can be grouped into two distinct classes: *periodic* and *aperiodic*. A periodic service is one that is required to be delivered repeatedly on a regular basis, for example every 50 milliseconds, every 10 seconds, etc. The timing constraints on such a service may correspond to the beginning and end of each period, or they may be given as explicit start and end times, in which case the service should be delivered within the appropriate interval in each period. Periodic services are common in a wide range of real-time applications, particularly those that are based upon the repeated application of well-defined control laws, and they possess the advantage that it is possible to calculate their worst-case processing requirements in advance, since both the period and maximum computation time per period must be known.

Aperiodic services are those that are required to be delivered at irregular intervals, typically as responses to events in a system's environment, and their timing constraints are usually given directly in terms of a liveline and a corresponding deadline. These might be expressed either as absolute time values according to some clock or as relative times, measured from the occurrence of the triggering event to which the service is responding. In principle, the worst case processing requirements for an aperiodic service cannot be calculated in advance, since there may be no limit on the number of simultaneous service requests that might be generated at a given time. However, in some cases, it can be assumed that a bound does exist on the number of simultaneous requests from the same source, allowing worst case figures to be obtained. Services that fall into this restricted aperiodic class are usually referred to as *sporadic* services.

2.1.3. Hard and Soft Services.

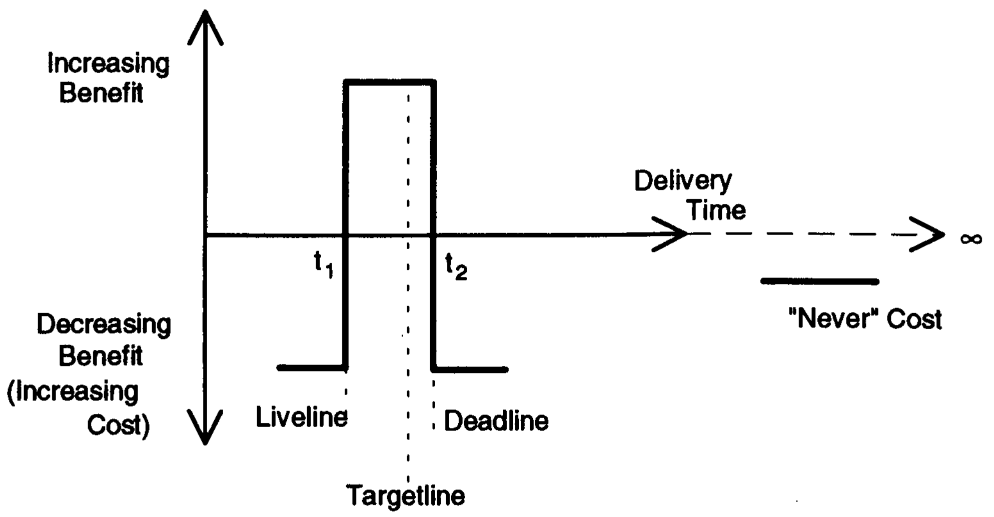


Figure 2.2. Hard real-time task with negative value outside timing constraint.

The other important property of any real-time service is the potential cost when it misses its timing constraint. In some cases, missing a deadline might carry no penalty, or it might only lead to a slight degradation in the performance of a system, however in other cases, the consequences of missing a timing constraint can be much more costly or damaging, both to the system and its environment. An analysis of such costs can only be carried out on a case by case basis, since different services will have different service requirements and failure modes, however it is possible to identify two broad classes of real-time service: *hard* and *soft.*[1] A *hard* real-time service has negative or zero utility if it is delivered outside its specified time interval, where negative utility represents some cost to the application or system. Figure 2.1 therefore represents a hard service, as does figure 2.2, which shows a service that has a negative utility outside its timing constraints. *Soft* real-time services have a positive, but sub-optimal, value to the system if delivered outside their specified timing constraint and only have a negative or zero value if delivered outside some wider time interval. This is illustrated in figure 2.3, which shows a soft real-time task that has increasing value between its liveline and its targetline, constant value between its targetline and its deadline and decreasing value thereafter.

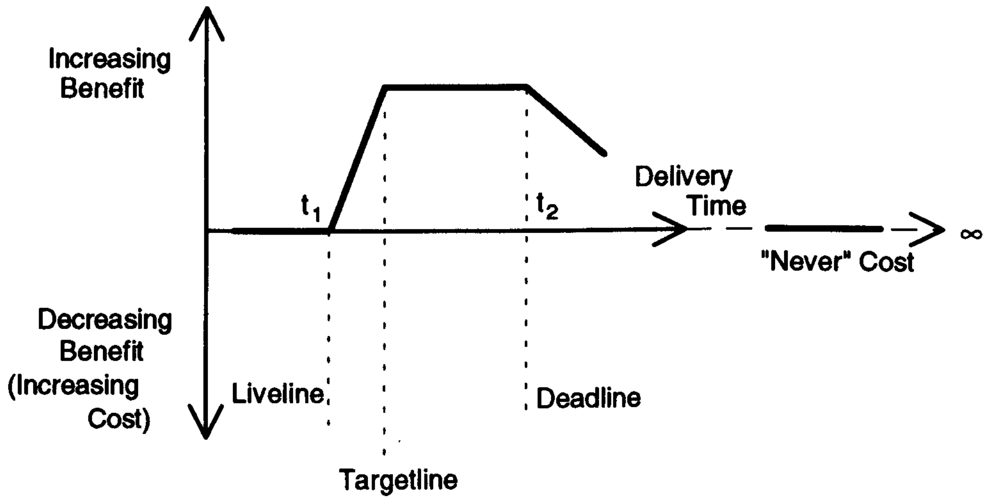


Figure 2.3. Value of soft real-time task vs. delivery time.

2.2. Real-Time Scheduling.

Finding feasible execution schedules for sets of real-time services is one of the central problems of real-time systems design ([Burns 88][Cheng 88]). A feasible schedule is one in which all services meet their timing constraints and the calculation of such a schedule can be extremely difficult. Real-time scheduling algorithms must not only take account of the

[1]The terms hard and soft as applied to individual real-time services should not be confused with the same terms as applied to entire real-time systems. However, there is a correspondence in their meanings.

processing requirements of each service, they must also take account of the timing constraints involved and the relative importance of different services for the continued operation of the system as a whole. This can give rise to situations where some sets of services cannot be scheduled at all because a system does not possess sufficient processing capacity to allow all timing constraints to be met.

2.2.1. Precedence and Priority.

In those cases where a set of services is such that it can be scheduled, there are a number of real-time scheduling algorithms than can be employed to find an appropriate, feasible schedule. The common requirement that all such algorithms must meet is that they have to be able to make correct decisions regarding which of a set of real-time service requests should be handled at any given time. In order to make such decisions, the scheduling algorithm must rely on some notion of *precedence*. The precedence of a real-time service can be defined as a generic representation of the necessary timeliness requirements of the service as perceived by a system designer [Bond 91]. In most cases, this can be taken to be some combination of the service's timing constraint (e.g. targetline or deadline) and its *priority*, which is a measure of the cost of the service missing its timing constraint. The priority of a service is an important parameter in the majority of real-time scheduling algorithms, since it expresses the importance of delivering the service on time. Hard services are therefore given high priority values while soft services are given lower ones.

2.2.2. Single Processor Scheduling.

Although there are several different real-time scheduling algorithms that can be used to schedule tasks on a single processor, they all fall into one of two basic classes. *Off-line* or *static* schedulers, such as the Rate Monotonic algorithm [Liu 73][Sha 86] and its derivatives, are used during system design and they produce fixed schedules for all anticipated service requests. The result is a timetable with slots reserved for all process executions and message transmissions. Events that require a change in the behaviour of the system at run-time are detected in a pre-determined slot and an alternative fixed schedule is installed. The fixed schedules also take account of dependencies between different services so that delays are not introduced. The great advantage of this approach is that it can give guaranteed performance and the schedulability of services can be assured. However, static scheduling is largely limited to periodic and sporadic services, since processing requirements need to be known or estimated in advance to generate the fixed schedules.

The other class of real-time schedulers does not suffer from this disadvantage, but it cannot offer the same degree of guaranteed performance as static scheduling. *On-line* or *dynamic* schedulers, such as the Earliest Deadline First (EDF) or Least Laxity First (LLF) algorithms, are executed at run-time when a system is in service. Using heuristics that must not themselves impose a significant system load, an appropriate schedule is generated for service requests as and when they arrive. Dependencies and deadlocks must either be avoided by the adoption of a suitable system design, or the resultant delays must be minimised either by the use of priority inheritance protocols [Babaoglu 90] or by the use of distributed deadlock detection mechanisms. The advantage of the dynamic approach is its flexibility, but the fact that it cannot give an advance guarantee of schedulability means that it is often regarded as being unsuitable for safety-critical applications or applications with strict timeliness requirements.

2.2.3. Scheduling with Resource Constraints.

The problem of scheduling a set of tasks on a single processor is further complicated when precedence or resource constraints are introduced. If two tasks must share some resource in a controlled manner (e.g. mutual exclusion) or if they must obey some precedence relationship (e.g. task A must execute before task B), any real-time scheduling algorithm that is used to schedule those tasks must not only take account of the timing constraints involved, but also the precedence or resource constraints. A simple application of the rate monotonic or earliest deadline first algorithms may not, therefore, be sufficient to produce a feasible schedule. In fact, it has been shown that the problem of deciding whether it is possible to schedule a set of periodic processes that use semaphores to enforce mutual exclusion is NP-hard [Mok 83]. However, in spite of such complexity, appropriate scheduling algorithms can be developed to find feasible schedules for particular classes of task or task set (for example, see [Blazewicz 79], [Zhao 87a], [Zhao 87b]).

2.2.4. Multi-Processor and Distributed Scheduling.

Just as the problem of scheduling a set of tasks with timing and mutual exclusion constraints is known to be NP-hard, so is the problem of finding an optimal schedule for a set of tasks on a multiprocessor system. Hence, the approach that is usually taken in such cases is to find some way to simplify the problem and produce an adequate, although sub-optimal, solution. Similarly, in distributed systems the problem of finding a feasible execution schedule for a set of real-time tasks can be very complex. In such systems, a two-level approach is often taken, with the tasks on each processing node being scheduled using a suitable single processor (or multiprocessor) scheduling algorithm, while a higher level, global scheduling strategy is employed to handle communications between tasks on different nodes and, where necessary, the allocation of tasks to nodes.

As with single processor scheduling under time and resource constraints, a number of scheduling algorithms have been developed for multiprocessor and distributed systems. For example, see [Muntz 70], [Ramamritham 90], [Chang 86], [Cheng 86] and [Ramamritham 84].

2.3. Supporting Fault Tolerance.

Another feature of the real-time applications domain is that real-time systems often have to meet reliability as well as timeliness requirements. Such reliability goals can be attained by adopting two complementary approaches. *Fault avoidance* techniques, such as formal specification and verification methods, can be used in an attempt to ensure that the system specification, design and implementation are as free from errors as is possible. *Fault tolerance* mechanisms can then be built into the system to ensure that, at run-time, it continues to provide an acceptable level of service even in the presence of a bounded number of internal faults.

Two techniques that have been widely used to support fault tolerance in programs are atomic actions (or transactions) and exception handling. Both of these can be regarded as program structuring techniques that provide a framework for error processing, in particular error recovery and, in the case of atomic actions, error containment.

2.3.1. Atomic Actions. (Transactions.)

In essence, an *atomic action* or *transaction* [Lomet 77][Lampson 81][Spector 83] is an "all or nothing" operation which either completes successfully (*commits*) having its intended effect or fails entirely (*aborts*) having no effect upon the state of the system. In more formal terms, the atomic action is said to possess the following three fundamental properties:

Failure Atomicity

Atomic actions either complete successfully having their intended effect upon the state of the system, or fail entirely having no effect at all.

Serialisability

The net effect of the concurrent execution of two or more atomic actions is equivalent to the net effect of some serial order of execution of those two actions. That is to say, there is no interference between parallel atomic actions.

Permanence of Effect

The effects of correctly terminated atomic actions are reflected in the system state and not lost as a result of subsequent errors or failures.

The first of these properties, failure atomicity, can be achieved by the use of an appropriate error recovery mechanism and this will often be based upon *backward* (*state-based*) error recovery. With a backward recovery scheme, aborting an atomic action causes the system to be restored to the state that it was in before the action began. This can be implemented using a checkpoint and recovery technique as follows. When an atomic action begins, a checkpoint of the current state of the system is taken. This could be a full checkpoint of all state information within the system or it might only include those parts of the system state that the atomic action can affect. Once this has been done, the action can be allowed to proceed. Later, if the action commits, any changes that it has made to the system state during its execution can be made permanent and the checkpoint discarded, however if the action aborts, any state changes that it has made can be ignored and the original system state restored from the checkpoint. This is, essentially, a version of the recovery block technique [Horning 74][Anderson 76] and it has become a popular implementation strategy.

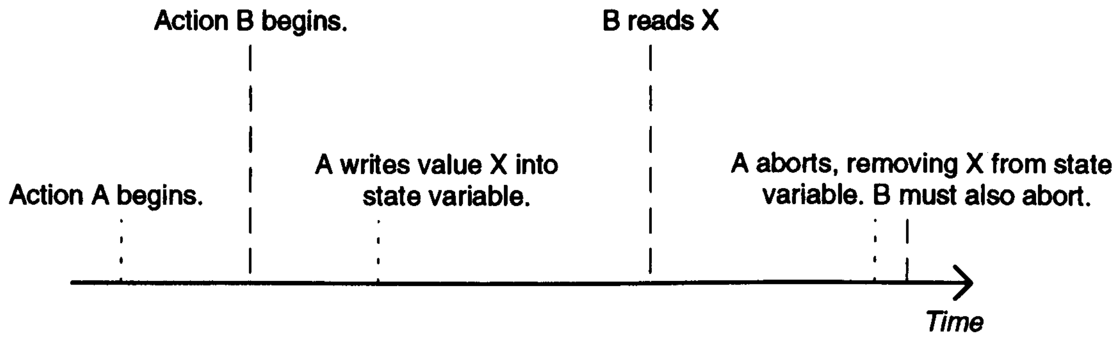

Action B begins.
B reads X
Action A begins.
A writes value X into state variable.
A aborts, removing X from state variable. B must also abort.
Time

Figure 2.4. Dependency between uncommitted actions.

The serialisability property of the atomic action is required to prevent dependencies forming between uncommitted, concurrent actions. For example, consider the situation illustrated in figure 2.4. An action, A, writes some value, X, into a state variable, from which it is read by another action, B. The action A then aborts, causing the value X to be removed. At this point, the action B must also be aborted, because it has "seen" X and may, therefore, have acted upon erroneous information. The situation can be further complicated if B has already committed when A aborts, since there may be other actions, dependent upon B, which must also be aborted. This is a classic example of the *cascading aborts*, or *cascade rollback*, problem and it should be avoided whenever possible. This is the purpose of the atomic action serialisability mechanism, since it prevents the formation of dependencies like the one described above. Methods that can be used to enforce serialisability include locking schemes in which actions lock those resources which they use until commit or abort time (e.g. [Eswaran 76]) and optimistic schemes [Kung 81], where some actions are allowed to complete and others forced to abort when conflicts occur. (For a more complete discussion of concurrency control for atomic actions, see for example [Parrington 88]).

The third property of the atomic action, permanence of effect, ensures that failures or errors do not lead to the loss of existing, committed results. This is usually implemented using some kind of stable storage medium (e.g. disk storage) and write-ahead logs, in which actions record their state updates and which are not made permanent (i.e. written to stable storage) until an action commits. Permanence of effect is particularly important in a distributed environment, where processing nodes can suffer independent failures. When a failed processing node is repaired or recovers and comes back on-line, its internal state will be consistent and the effects of any committed atomic actions that were run on that node before it failed will be preserved. This minimises the time required for the node to be brought up to date with activities in the rest of the system.

2.3.1.1. Nesting and Top-level Actions.

Atomic actions, like any other block-structured programming construct, can be nested as shown in figure 2.5. Nested actions behave in the same way as any other atomic action with respect to error recovery, however when a nested action commits, any resources that it was holding (e.g. any locks that it held) are not released. Instead, they are passed on to the enclosing parent action. This is to allow for the situation in which the parent action subsequently aborts, since it must then be able to recover the effects of those nested actions that have committed.

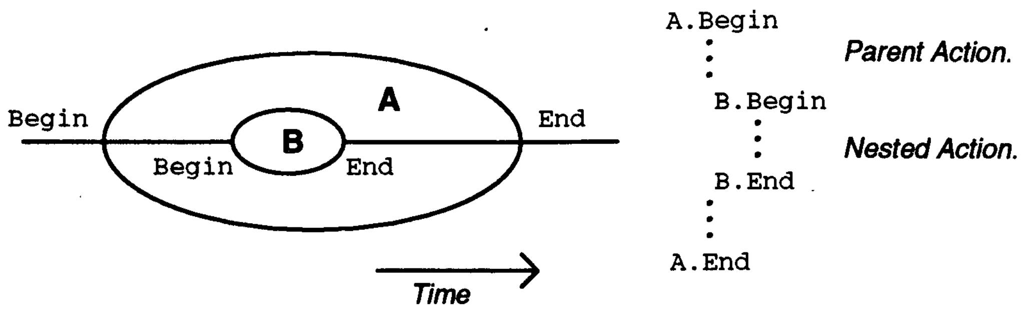


Figure 2.5. Nested Atomic Actions.

In some cases, an atomic action may be logically nested, but its effects upon the state of the system once it has committed should not be recovered, even if its parent action aborts. Such an action is known as a ***nested top-level action***, since it is regarded as being at the top (i.e. outermost) level of the nesting hierarchy for the purposes of error recovery (see figure 2.6). Top-level actions can be useful in a number of situations, particularly where an action is capable of causing external effects that cannot be recovered using the normal backward recovery mechanisms.

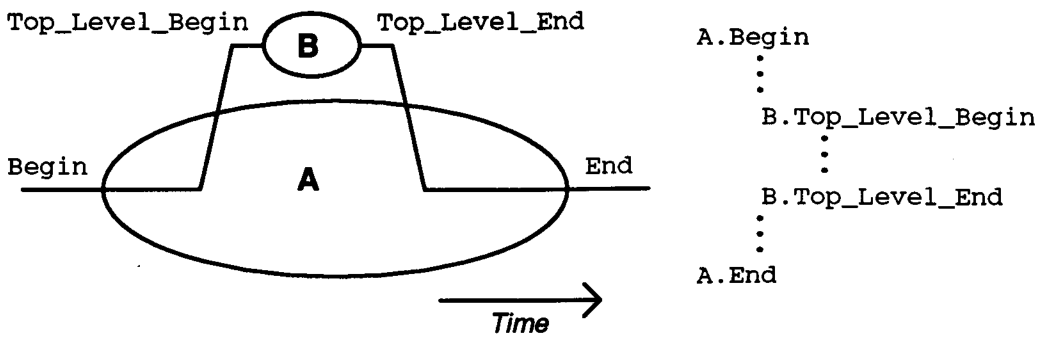


Figure 2.6. Nested top-level atomic action.

2.3.1.2. Atomic Actions : Advantages and Disadvantages.

Generally, atomic actions provide a good structuring technique for distributed application programs and their use helps to ensure that application state information remains consistent even in the presence of errors or failures. The abort mechanism of the atomic action provides error recovery and damage containment is, to some extent, provided through serialisability. A wide range of faults can be handled, including unanticipated ones, so long as the integrity of the atomic action mechanism itself is not compromised and, for general classes of applications, atomic actions can be a very useful fault tolerance technique. However, in real-time applications, the use of an atomic action mechanism can present problems. First and foremost, the use of backward error recovery techniques (i.e. state restoration) to provide error recovery may be inappropriate. In particular, while the processes of checkpointing and restoring state can themselves be made both fast and efficient, the subsequent re-execution of an action or some alternative operation may be time-consuming and cause deadlines to be missed. Furthermore, real-time systems have to interface to an external environment and it is possible that the environment will not be able to carry out any kind of state restoration. For instance, if some operation in a chemical plant control system causes 100 litres of reactant Z to be flushed into a fully charged reaction vessel and that operation is subsequently found to be in error, then there is no way in which those 100 litres of reactant Z can be removed from the vessel other than emptying its entire contents. That is to say, there is no direct way to restore the original state of the system and alternative recovery techniques must be used to provide the abstraction of failure atomicity.

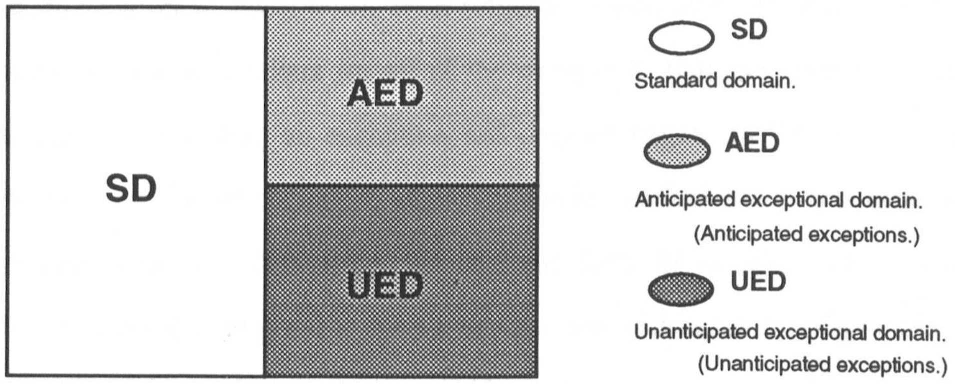
Another potential problem area with atomic actions in the real-time domain is the requirement for serialisability. The majority of the mechanisms presently used to enforce serialisability take no account of the timing constraints or criticality of different actions. Consequently, tasks may be delayed arbitrarily while waiting for locks or aborted randomly under optimistic schemes when conflicts occur. This indiscriminate treatment of potentially critical tasks would be totally inappropriate in a real-time system, where the criticality of different tasks and their different timing constraints must be taken into account. However, other concurrency control mechanisms have been developed for use with transactions in real-time databases (see, for example, [Stankovic 88b][Sha 88][Sha 91]) and such alternative mechanisms could easily be used to enforce serialisability in more general classes of transaction-based system.

2.3.2. Exception Handling.

Another fault tolerance technique which is well-known and which has been widely used in a number of systems is exception handling [Goodenough 75][Cristian 82]. Like atomic actions, exception handling can be viewed as a program structuring technique that provides support for error recovery, however where atomic actions provide a generic recovery mechanism, exception handling is used to provide specific recovery operations for specific, anticipated errors.

2.3.2.1. Anticipated and Unanticipated Exceptions.

Given some operation, which is intended to provide a particular service, the set of possible inputs to that operation can be divided into two disjoint domains. The *standard domain* of inputs contains those input values that lead to a correct output, while the *exceptional domain* contains input values that cause an erroneous output to be produced. Values falling in the exceptional input domain include those for which it is known that the operation cannot provide a correct output (for example, negative inputs to a square root function) and those for which the operation should return a correct output but, instead, returns an incorrect one due to the existence of an internal fault. The difference between these two types of exceptional input is of particular interest. Input values for which an output is not defined or for which it is known that a correct output cannot be produced, can be anticipated and *exceptions* declared for them. Then, when an operation is invoked with one of these input values, it can raise the appropriate exception to its caller. This signals to the caller that correct service cannot be provided and the caller can then provide a corresponding *exception handler* to perform appropriate error recovery. Thus, for any operation it is possible to identify an *anticipated exceptional domain*, for which exceptions are defined and an *unanticipated exceptional domain* (see Figure 2.7).



SD
AED
UED
SD
Standard domain.
AED
Anticipated exceptional domain.
(Anticipated exceptions.)
UED
Unanticipated exceptional domain.
(Unanticipated exceptions.)

Input in :	Results:
SD	Correct outputs.
AED	Exceptional response.
UED	Error or failure.

Figure 2.7. Standard and Exceptional Input Domains.

2.3.2.2. Termination vs. Resumption.

When designing an exception handling mechanism, there are three basic issues that must be taken into consideration. The first of these is whether to adopt the *termination model* of exception handling or the *resumption model*. In the former, when an exception is raised, execution of the current operation is terminated and control returns to its caller where the exception is handled. In the latter model, when an exception is raised, control returns to the caller of the current operation for the exception to be handled, but execution then resumes from the point in the current operation at which the exception was raised. In many cases, there are strong arguments for adopting the termination model, since the detection of an exception means, by definition, that a standard (correct) service cannot be provided so it will often be more appropriate to terminate the current operation entirely, rather than attempt to handle the exception and then resume the operation. An exception handling mechanism based upon the termination model is also likely to be somewhat easier to implement than one based on the resumption model.

2.3.2.3. Single-Level vs. Multi-Level.

The second thing to consider when designing an exception handling mechanism is whether it should be *single-level* or *multi-level*. With single-level exception handling, the caller of an operation must provide handlers for all of the exceptions that the operation might signal during its execution. Hence, an exception will only propagate as far as the scope of the immediate caller of a failed operation, where it must be handled either by a specific declared exception handler or by a default handler of some kind. In the event of the caller of an operation not being able to provide such a handler and wishing to propagate the exception to higher levels of the system, it must explicitly declare and raise that exception itself. This is as opposed to multi-level exception handling where exceptions are allowed to propagate back up the call stack until an appropriate handler is found. The net effect of this behaviour is that exceptions raised by some operation X may be handled by any of the modules or procedures higher than X in the current stack. For example, some exceptions may be handled by the immediate caller of operation X, while others are handled by the caller of that module and still others are handled by the caller of that module and so on.

Single-level exception handling mechanisms are usually favoured because they can be easier to implement than a multi-level mechanism. It can also be argued that they enforce good programming practice when using exceptions, since exceptions must be handled at the point at which they are initially detected and the propagation of exceptions to higher levels of the system is made explicit.

2.3.2.4. Parameter Passing.

The final decision to be made in the design of an exception handling scheme is whether or not to allow parameters to be passed back to exception handlers when exceptions are raised. The argument in favour of parameter passing is that it can be used give exception handlers access to more specific state information about the operation that raised the exception, as well as data regarding the exception itself. This, in turn, allows an exception handler to select the most appropriate form of error recovery for the current situation. However, introducing parameter passing into an exception handling mechanism complicates its implementation and, in the majority of existing systems that provide some kind of exception handling, parameter passing is not supported.

2.3.2.5. Exception Handling : Use and Advantages.

The inclusion of exception handling mechanisms as part of the native functionality of programming languages has become common and several existing languages, including CLU [Liskov 79], Mesa [Mitchell 79], Ada [Ada 80] and recent versions of C++ [Stroustrup 87][Koenig 90], provide support for exceptions. For example, in the CLU language, a single level, termination mechanism was provided as shown in Figure 2.8.

```
div =   proc(x,y:int) returns (int) signals (divide_by_zero)
        if y=0 then
          signal divide_by_zero
        else
          return (x/y)
        end div
```

```
z = div(a,b)
        except when divide_by_zero:
                  z = 0
        end
```

Figure 2.8. Exception Handling in the CLU Language.

The major advantage of exception handling is that it allows code for normal and exceptional processing to be logically separated, thus providing a flexible and powerful, yet manageable, error recovery mechanism for anticipated errors. Exception handling can also be used to provide fault tolerance coverage for unanticipated errors. This can be achieved either by extending the exception handling mechanism itself to permit the definition of default exception handlers that implement some kind of generic recovery strategy, or by using exception handling in conjunction with some other fault tolerance technique, such as atomic actions, that is capable of handling those errors for which exceptions have not been defined.

For all practical purposes, the use of exception handling does not have any major disadvantages. However, an important point to note is that, in real terms, exception handling only provides a mechanism for error recovery. Processes such as error detection, error containment and fault treatment are then handled explicitly by means of defining an appropriate set of exceptions and providing appropriate handlers for them.

2.4. Some Existing Systems.

Before moving on to consider the problems associated with the use of replication to provide fault tolerance in real-time applications, it will be worthwhile to examine some existing fault tolerant real-time systems. Although there are several such systems currently in use or under development, for example MARS, ARTS [Tokuda 89], CHAOS, MARUTI, SPRING [Stankovic 89] and Delta-4 XPA [Barrett 90], only three - specifically MARS, CHAOS and MARUTI - will be considered here. All three of these systems are intended for use in a real-time environment and they each incorporate some form of fault tolerance. Furthermore, these three systems provide good examples of both the static (MARS) and dynamic (CHAOS) approaches to real-time applications, as well as showing a way in which the two approaches can be, to some extent, combined (MARUTI).

2.4.1. The MARS System.

The MARS (Maintainable Real-time System) [Kopetz 89][Damm 89] project has now been underway for over a decade, initially at the Technical University of Berlin, but now at the Technical University of Vienna. A first prototype of the system was built in 1984 and the second prototype has been operational since 1988.

MARS is a fault tolerant, distributed real-time systems architecture intended for use in closed loop (sensor - control system - actuator) process control applications. In order to provide guaranteed levels of service, MARS is a completely static system with task execution schedules and communications schedules all computed off-line and held in tables which are consulted by the MARS operating system at run-time.

2.4.1.1. MARS Design and Computational Model.

The MARS design approach is based upon offering deterministic behaviour even when the system is operating under peak load conditions. Since MARS is capable of meeting hard real-time constraints under peak load, it can naturally accommodate low-load conditions as well. MARS applications are based upon a transaction model in which sequences of inter-related actions transfer the system from one consistent state to another. These actions may themselves be made up of simpler, more primitive actions. An action is triggered by some specified stimulus and produces some kind of response. If that response must be generated within a given interval after the stimulus occurs, the action is regarded as being a real-time transaction.

A MARS system configuration consists of a set of clusters (*MARS clusters*), each of which is composed of several components interconnected by a synchronous real-time bus. An individual component is a self-contained computer, including application software, and it represents a combined hardware and software unit offering some given functionality and performance. Each component will execute a set of real-time tasks and an identical copy of the MARS operating system kernel.

Tasks and components communicate by means of *state messages*. Conceptually, these are similar to global variables in a programming language. Once produced, they are read-only, but they can be read an arbitrary number of times by an arbitrary number of tasks. They are used to exchange information about the state of the environment or some internal state of the system itself. Each state message has a *validity time*, after which it will be discarded by the system, and a cluster-wide unique name which refers to both its data type and its actual semantic content (i.e. what its value represents at the application level). Only one message with a given name can be valid at any given time so, typically, more recent messages in a sequence supersede the older ones.

The use of state messages in this way means that the number of message buffers used is static and tasks can allocate an appropriate amount of buffer space based upon the messages that they will generate or receive. Buffer management is then handled by the operating system at run-time. Communication time over the MARS bus is pre-scheduled using a *TDMA* (Time-Division Multiple Access) policy in which each task is given a specified set of communication slots for its own dedicated use. This avoids problems with communication conflicts, while the periodic nature of the system and the semantics of the state message mechanism mean that explicit flow control is not required.

2.4.1.2. Scheduling and Real-Time Tasks in MARS.

MARS supports both hard and soft real-time tasks, the former being pre-scheduled according to their worst-case execution times, while the latter execute in the spare processing time that is available when hard real-time tasks complete early. Since the strictly periodic nature of the MARS system makes dynamic scheduling unnecessary, all task scheduling is performed off-line, taking into account the maximum execution times of tasks, their co-operation by message exchange and the assignment of messages to TDMA slots. Tables produced by the off-line scheduler give the start and end times of tasks and they are linked into the core image of each individual component. At run-time, task switching is performed within the MARS kernel's clock interrupt handler according to the scheduling tables and tasks must release the CPU (using a *suspend* system call) before their specified end time otherwise the kernel will regard it as an error (time-limit exceeded). To provide some degree of flexibility, several schedules may be calculated, defining different operating modes for the system. Switching between schedules is caused either by an explicit schedule switch request from an application task or by the receipt of a message associated with a schedule switch. Two different types of schedule switch can be performed. One is a "clean" switch which guarantees the preservation of consistency within the application, but which can only be performed at certain times defined within the design of the application itself. The other is a much faster switch which does not guarantee consistency, but which takes effect at the next invocation of the major clock interrupt handler (every 8ms). This fast schedule switch mechanism is purely intended for emergency situations where some form of catastrophic failure must be avoided. The importance of the clock interrupt handler in the above task and schedule switching mechanisms is due to fact that the clock interrupt, coming from the clock synchronisation unit, is the only interrupt that is allowed in the MARS system. All other interrupt handling, including device drivers, must be performed by low-level routines which are themselves initiated by the clock interrupt routine.

2.4.1.3. MARS Clock Synchronisation.

As well as providing the clock interrupt to the processor, the *clock synchronisation unit* (*CSU*) [Kopetz 87] co-operates with an operating system synchronisation task to synchronise clocks between individual MARS components and MARS clusters. Whenever a message is sent, the CSU provides a *sender time-stamp* to the network controller chip which adds it to the outbound message. Then, whenever a message arrives, the network controller chip generates an interrupt to the CSU to obtain a *receiver time-stamp* for the message. These time-stamps are used to perform clock synchronisation using a fault tolerant average algorithm. Synchronisation to an external time standard (International Atomic Time) is provided from a radio source using signal modulation and processing techniques and, in this case, a rate correction is broadcast to all clocks independently of the internal, time-stamp based, synchronisation.

2.4.1.4. Fault Tolerance in MARS.

Finally, fault tolerance in MARS is provided by using extensive self-checking in individual components in conjunction with active replication. High error detection coverage is provided by the use of software error detection mechanisms at the operating system level and hardware mechanisms inherent in the processors themselves. MARS components therefore provide, to a high degree, the abstraction of *fail-silence*. That is to say, when a component suffers a failure, it ceases to produce outputs. Active replication of such components does not require a voting mechanism, since failed replicas will not respond to requests and the first response generated by any member of the replica group can be assumed to be correct. This replication scheme provides an effective fault tolerance mechanism, however it also imposes the restriction that computations must be deterministic, otherwise the internal states of different replicas may diverge and inconsistent responses may be received from the replica group if replies are received from different replicas at different times. This is an unavoidable consequence of using active replication and it is not unique to the MARS system implementation. Tolerance of communication faults is achieved by sending all messages several times over duplicated real-time busses. The semantics of the MARS state messages makes this a feasible and easily implemented fault tolerance strategy, offering a high probability that components will receive at least one good copy of any message.

2.4.2. CHAOS.

The CHAOS (Concurrent Hierarchical Adaptable Operating System) project [Schwan 90a] [Gopinath 89] is intended to support the programming of real-time applications that are:

- *Efficient* where efficiency is loosely defined as "not significant system overhead"
- *Accountable* At all times applications should attempt to meet specifications imposed by the applications programmer

- *Predictable* Given similar sets of events, an application should respond similarly in all cases.

To achieve these properties, CHAOS combines an object-based programming and execution model [Schwan 87][Gheith 90] with a rich set of invocation primitives and a set of adaptation mechanisms which facilitate both *static* and *dynamic* adaptation of application software. Like MARS, CHAOS provides support for real-time transactions, but unlike MARS CHAOS is a highly dynamic system which uses on-line task scheduling.

2.4.2.1. CHAOS Application Structure.

CHAOS applications are structured as sets of autonomous objects interacting by means of operation invocations. Objects may be *active* (i.e. contain internal threads of control and undertake activity independently of incoming invocations) with a single co-ordinating process which accepts invocations and schedules them for execution by one of the object's server threads. Concurrency control within an object is provided either by the co-ordinator process or by the use of real-time locks. These provide ordinary *mutex* (mutual exclusion) locking, but with an associated timeout which prevents a thread holding locks for an unbounded time and which circumvents the problems associated with unbalanced lock and unlock instructions.

2.4.2.2. Invocations in CHAOS.

In many respects, the strength of CHAOS lies not in its object model, but within its sophisticated range of invocation primitives. There are three basic classes of invocation, giving the choice of a high-speed control invocation with no transfer of data, a stream invocation which provides dedicated data transfer after an initial control phase establishes a connection and a normal method invocation involving transfer of both control and data. Any of these invocation types may be used synchronously or asynchronously and any invocation may have explicit real-time constraints or scheduling parameters associated with it as well as real-time attributes such as criticality. Sporadic, periodic or event-driven invocations are supported and four different classes of real-time deadline are recognised:

> *Guaranteed deadlines* which must be met under all circumstances
>
> *Soft deadlines* which may occasionally be missed
>
> *Weak deadlines* which allow tasks to return partial or incomplete results
>
> *Recoverable deadlines* which cause programmed recovery actions to be taken if they are missed.

When an invocation is made, much of the cost is borne by the invoked object, making invocation requests cheap for the invoker. Further, if an appropriate mechanism is not provided by CHAOS, the applications programmer can synthesise an appropriate invocation primitive from a number of basic invocation building blocks. Such synthesised invocations incur a slightly greater overhead than the optimised primitives provided by the system, however this may still be preferable to using an inappropriate built-in invocation. This invocation synthesis mechanism is one of the main static (i.e. fixed at compile time) adaptation mechanism provided by CHAOS and it allows individual CHAOS applications to be optimised for their particular environment.

2.4.2.3. CHAOS and Atomic Transactions.

CHAOS supports an atomic transaction model in which transactions are taken to be groups of related invocations that are to be guaranteed as a single execution unit. These atomic, real-time computations (as they are usually called in CHAOS terms) possess three classes of attribute. *Real-time* attributes specify temporal restrictions on the execution of the computation (e.g. start times and deadlines). *Concurrency control* attributes constrain the execution of concurrent atomic computations in order to control access to shared resources and to maintain serialisability. *Recovery* attributes are application dependent properties required to guarantee that aborted computations leave the system in a consistent state.

Unlike MARS, in which real-time transactions are largely used as a structuring technique for applications and fault tolerance is supported using active replication, the transaction mechanisms in CHAOS are intended to provide both an application structuring technique and fault tolerance. Application specific recovery for aborted computations is provided by the use of *anti-operations*. These are private operations within objects which implement appropriate recovery actions for the objects' normal, public operations. In some cases, an anti-operation may simply provide backward recovery as with a classical transaction mechanism, however anti-operations may also provide some form of compensation or forward recovery when it is needed.

2.4.2.4. Scheduling in CHAOS.

As mentioned earlier, CHAOS is a dynamic system in the sense that all schedulability analysis and task scheduling is carried out at run-time, taking account of task timing constraints and criticality. An optimised implementation of the Earliest Deadline First algorithm (see [Schwan 90b]) is used for all thread scheduling within CHAOS objects. Naturally, this approach cannot hope to offer the guaranteed service or response times of MARS and the overhead of scheduling at run-time must itself be included in the on-line schedulability analysis. However, this makes for a highly flexible system which is capable of coping with highly dynamic and unpredictable operating environments. Dynamic adaptation mechanisms that re-compute task deadlines or introduce lower functionality, higher performance versions of objects at times of overload further help to ensure that the most critical application tasks continue to be executed even under peak loading. Consequently, CHAOS can handle a wide range of applications and operating conditions.

2.4.3. MARUTI.

The MARUTI project [Agrawala 89][Levi 89][Mosse 91b] at the University of Maryland has studied the problems associated with the design, development and deployment of distributed real-time applications which must meet specified security and dependability requirements and which can be implemented on a heterogeneous, multi-machine platform.

In essence, MARUTI represents an attempt to develop an integrated system according to a comprehensive integrated development methodology and a prototype system has already been implemented (on top of UNIX) to demonstrate the feasibility of this approach.

2.4.3.1. MARUTI Objects.

Like CHAOS, MARUTI is an object-based system in which the concept of objects has been applied at all levels of the system itself. Interactions between objects are regarded as being service requests in which the caller requests some service from the callee through a specified *Service Access Point* (*SAP*). Applications and application tasks can then be viewed as a rooted, directed graph (referred to as a *computation graph*) in which vertices represent services and edges represent the invocation messages (possibly containing data) sent from one service to another. This view corresponds directly to the usual notion of objects interacting by means of invoking one another's methods, although in MARUTI the basic model of invocation is one way (asynchronous) rather than the more common synchronous remote procedure call (*RPC*).

A novel feature of MARUTI objects is the inclusion of an extra control part (called a *joint*) within each object in addition to its usual state information and its methods. The joint is basically an auxiliary data structure which maintains certain static and dynamic data about the object itself. For example:

- User and owner information
- Resource and/or server requirements
- A ticket check mechanism (for protection and security purposes)
- Timing Constraints
- A replica/alternative control mechanism (for fault tolerance)

This extension to the normal object model makes it easier for MARUTI to provide support for hard, real-time computations, particularly with regard to achieving bounds and guarantees for task execution times.

2.4.3.2. Scheduling in MARUTI.

Task scheduling in MARUTI is viewed as a resource allocation problem, where the resource in question is execution time on a processor. When a job is submitted to the system, the timing constraints for this execution are specified, typically in the form of a ready time (earliest start time) and a deadline. Resource allocation for the new job then takes place, including a schedule feasibility check to ensure that there exists an execution schedule including the new job in which all tasks (the new one and any existing, guaranteed jobs) can meet their timing constraints. The schedulability check takes into account local processor scheduling for a given object, as well as communication time and remote server scheduling where necessary.

The actual schedulability check which is used in the MARUTI system is based upon a formal, interval-based notion of time and it works, in simple terms, by trying to find a window (or set of windows) of execution time that will allow the new task to be executed within its timing constraint, but without jeopardising the execution of any other task which has already been accepted. In order to find such a window, the execution of existing tasks may have to be re-scheduled, so long as it does not cause the violation of a timing constraint. The precise sequence of events that takes place on submission of an execution request is as follows. When it arrives at an object, an incoming invocation's timing constraint is tested for insertion into the object's *calendar* (an auxiliary data structure in the object's joint holding a schedule for accepted requests). Assuming that the request can be scheduled, its timing constraint is inserted into the calendar and its computation interval is reserved. The reservation will expire after some time if an acknowledgement is not received from the initiator of the invocation request. This mechanism is required to handle those cases where a task involves a number of requests which must all be guaranteed - the acknowledgement will then only be sent if all invocations are accepted. In the event of an invocation request failing, the initiator of the request is informed and it becomes the initiator's responsibility to ensure that any successful reservations made elsewhere for the failed request, or related ones, are cancelled.

At first glance, it might appear that scheduling in MARUTI is dynamic as in the CHAOS system, however while execution requests for tasks are indeed submitted and scheduled at run-time, MARUTI scheduling is actually more like static scheduling since the computation graph for a job must be prepared in advance, including information such as resource and server requirements. Further, when an execution request is refused, no account is taken of the priority or criticality of the refused task. MARUTI therefore guarantees deadlines on a first come first served basis and if a highly critical task is to be guaranteed in its execution, reservations must be made in advance in the appropriate object calendars during application design. Hence, MARUTI scheduling has more in common with the static scheduling mechanisms of MARS than the dynamic EDF scheduling of CHAOS.

2.4.3.3. Fault Tolerance in MARUTI.

Fault tolerance in the MARUTI system is supported by means of replicating an application or task computation graph in separate, failure independent partitions of the distributed system on which it is to run [Mosse 91a][Zoubeir 91]. Only hardware faults such as power supply, processor or communication link failures are considered and it is assumed that all faults manifest themselves in the form of incorrect, late or missing messages. It is also assumed that the underlying system is fully connected when a job is submitted to guarantee that each node hosting a replica of the job will still be reachable even if a number of communication link failures equal to the task's resiliency level occur. This ensures that votes can be received from all replicas.

The underlying support for replication resides in the MARUTI communications system which provides *forkers* and *joiners* between the service access points of different replicas. A forker delivers a request to all members of a replica group, while joiners gather up requests from replicas and vote upon them before delivering the request to the appropriate service access point if it proves to be correct. The precise voting policy to be used by a joiner can be specified by the applications programmer and an appropriate number of copies of the job will be allocated by the system according to the voting criteria and required resiliency level.

2.4.3.4. The MARUTI Programming Language.

A final interesting feature of the MARUTI system is the existence of *MPL* (the MARUTI Programming Language) which has been specifically designed to allow the applications programmer access to the features of the MARUTI system [Nirkhe 90]. MPL is an object-oriented, statically-typed language which includes a range of real-time language constructs including block structured timing constraints, real-time synchronisation primitives and explicit loosely coupled or tightly coupled parallelism. MPL also offers an object group management mechanism which interfaces to the underlying system to provide complete transparency for issued such as naming, message transfer and object management.

2.5. MARS, CHAOS and MARUTI : A Brief Evaluation.

The three systems that have been described are all complete real-time operating systems in their own right and each one has been designed with some specific purpose in mind. In the MARS project, the major design goal was a drive for guaranteed, predictable performance at all times, leading to a static system with comprehensive pre-scheduling and strictly controlled run-time support mechanisms. The CHAOS system, on the other hand, was designed to support adaptable real-time applications that can modify their behaviour in the face of an unpredictable operating environment. This has led to a highly dynamic system that uses techniques such as on-line scheduling and that includes specific adaptation mechanisms within the operating system itself. Finally, the view taken in the design of the MARUTI system is that the application development process, as a whole, should be supported. Hence, the scope of MARUTI ranges from a methodology for application design and implementation to the provision of appropriate operating support mechanisms for that methodology.

Given the different goals and design approaches adopted in the three systems, it is only natural that each system will have its own specific advantages and disadvantages. It will, therefore, be interesting to consider this aspect of the MARS, CHAOS and MARUTI systems, giving a brief summary of the potential strengths and weaknesses of each approach.

2.5.1. MARS : Advantages and Disadvantages.

For hard real-time systems that must meet high dependability requirements, the MARS approach offers several advantages. The use of static scheduling both for tasks and for the communication between them makes it easy to reason about the run-time behaviour of the system under different operating conditions. Also, the use of state messages for communication avoids a number of problems (e.g. buffer overflows) and makes it easy to provide fault tolerance simply by replicating components within the system. The provision of fault tolerance is further simplified by the self-checking nature of MARS components, since further error detection and voting mechanisms do not have to be provided.

Unfortunately, some of these strengths in the MARS system can just as easily be regarded as potential weaknesses. Static scheduling requires tasks to be designed in such a way that maximum execution times can be calculated in advance, although this is true for any hard real-time system. More importantly, the use of static scheduling makes it difficult for the MARS system to adapt to dynamic or changing environments and complicates the problems associated with event handling. Further, the state message mechanism provides a representation of the state of a task rather than explicitly providing inter-task communication and the semantics of state messages can make producer-consumer or client-server relationships difficult to model. This has the effect of constraining the development of MARS applications, possibly to an undesirable extent in some cases.

2.5.2. CHAOS : Advantages and Disadvantages.

In many respects, the CHAOS system lies at the opposite end of the real-time spectrum to the MARS system. The great strength of CHAOS lies in its dynamic structure and its ability to adapt to external circumstances. This not only simplifies event handling, it also allows the CHAOS system to tolerate transient overload conditions and, where necessary, to provide for a graceful degradation of service in the face of failures.

However, considering disadvantages, the dynamic nature of the CHAOS system makes it difficult to reason about the run-time behaviour of the system or any applications running upon it. Although a great effort has been made to ensure as predictable and reliable a response as possible from the system, the use of on-line scheduling means that advance guarantees of schedulability, performance or correctness may be very difficult to obtain and the overheads of the scheduling algorithms that are used must be borne by the system at run-time. This further complicates the problem of analysing the run-time behaviour of CHAOS applications.

2.5.3. MARUTI : Advantages and Disadvantages.

If CHAOS and MARS represent two ends of the spectrum of real-time systems, MARUTI can largely be viewed as occupying the middle ground. Basically, it is a static system with pre-scheduling and pre-allocation of resources and yet, at the same time, it has been designed to support reactive real-time applications, with execution requests being generated at run-time in response to external stimuli. The scheduling and schedulability analysis mechanisms used in MARUTI are widely applicable to the real-time applications domain and the MARUTI view of real-time objects and application development also possesses some advantages, being both clear and well-structured.

On the other hand, MARUTI does have its weaknesses. The approach taken to fault tolerance may give cause for concern, since it is reliant upon being able to partition a distributed system into k failure-independent partitions, each of which must be capable of supporting an instance of the computation graph for a replicated task. The fault tolerance mechanisms also make no allowance for non-deterministic computations. (The importance of this latter topic will become clearer in the following chapter.) Finally, MARUTI scheduling does have one or two unfortunate features. If an execution request is submitted that cannot be scheduled, then it is rejected regardless of the importance of the task. This makes it necessary to design applications very carefully, making appropriate reservations within the system for all critical tasks, even if they are only executed in exceptional circumstances. Also, it is assumed that the network of machines is completely connected whenever an execution request is submitted and this may not always be true in a system in the presence of failures.

2.6. Chapter Summary.

This chapter has presented a review of some of the basic concepts in real-time and fault tolerant computing. The concept of a real-time service was introduced and discussed in some detail, including an examination of timing constraints and the notions of hard and soft as well as periodic and aperiodic services. A short summary of the real-time scheduling problem was then given, covering the basic concepts of precedence and priority and briefly discussing single processor, multi-processor and distributed scheduling algorithms.

Turning to fault tolerance, both atomic actions and exception handling were considered in detail. In the former case, this included a description of the basic properties of atomic actions along with a discussion of nested and nested top-level actions, while the description of exception handling covered both the termination and resumption models, as well as considering single-level versus multi-level mechanisms and the use of parameter passing in exceptions.

Finally, the chapter concluded with an examination of three existing fault tolerant real-time operating systems; MARS, CHAOS and MARUTI, and a brief evaluation of their relative strengths and weaknesses.

Chapter 3.

Object Replication in Real-Time Systems.

The use of replicated hardware or software components to provide fault tolerant services is a well-known technique that has been used in a number of different systems. Essentially, replication schemes provide the users of a service with the notion of an abstract component that exhibits the properties of a single component but that is actually made up of many replicas. The abstract component is then capable of continuing to provide normal service in the presence of some bounded number of replica failures. In the hardware case, several identical components (or non-identical ones designed to provide the same service) may be used to provide the abstraction of fault tolerant hardware. Similarly, by replicating software components (processes, objects, etc.) on different processing nodes in a distributed system, the abstraction of a fault tolerant software component can be provided. This abstract software component can be constructed in such a way as to tolerate faults in the underlying system hardware (processors, network, communications links, etc.) and, if necessary, faults in the individual software replicas themselves (by providing diverse implementations of the same service - i.e. N-version programming [Chen 78]).

In this chapter, and throughout the remainder of this thesis, it is the replication of software components to tolerate underlying hardware faults that will be the major area of interest. The abstract fault tolerant component can be represented by a *replica group*, which is a group of replicas co-operating to provide the same service. All interactions with such a replica group must be controlled by an appropriate *replication protocol*, in order to preserve the notion of the abstract component despite failures in individual replicas (see [Birman 88]). The replication protocol is instrumental in masking failures so that the replicated service can continue to function and, since there are several different types of failure that can occur in a distributed system, there are several different replication protocols that can be used. In any given case, the complexity of the protocol that is required depends upon the type of faulty behaviour that is to be tolerated, so we will begin with a brief classification of fault types, before moving on to discuss the replication protocols required to deal with them.

3.1. Fault and Failure Classification.

Each component within a system will have associated with it a specification of its correct behaviour for a given set of inputs. A *non-faulty* component will produce an output that is in accordance with this specification, while the response from a *faulty* component need not be specified and can, in principle, be anything. Adopting the view of [Shrivastava 90], the response from a component for a given input will only be considered to be correct if the output value is correct and if it is produced on time (i.e. within a specified time limit). This strictly defines correct behaviour for the component and it is then possible to classify the behaviour of a faulty component according to the way in which the component deviates from this general specification. Four possible classifications are:

Omission Fault/Failure.

A component that fails by not producing any output is exhibiting an *omission failure* and the corresponding fault is an *omission fault.* A communication link that occasionally loses messages is an example of a component suffering from an omission failure.

Value Fault/Failure.

A fault that causes a component to respond within the correct time interval, but with an incorrect value is termed a *value fault* (with the corresponding failure called a *value failure*). A communication link that delivers corrupted messages on time suffers from a value fault.

Timing Fault/Failure.

A *timing fault* causes a component to respond with the correct value but outside the specified time interval (either too early, or too late). The corresponding failure is a *timing failure.* An overloaded processor that produces correct values, but with an excessive delay suffers from a timing fault. Note that timing faults and timing failures can only occur in systems where timing constraints have been imposed on components' responses.

Arbitrary Fault/Failure.

The previous failure classes have specified how a component can be considered to fail in either the value domain or the time domain. It is also possible for a component to fail in both domains in a manner which is not covered by any of the previous classes. A failed component which produces such an output will be said to be exhibiting an *arbitrary* or, as it is sometimes known, *Byzantine* failure (after [Lamport 82]).

The last of these, an arbitrary fault can cause any unexpected violation of a component's specified behaviour, while all other fault classes preclude certain types of faulty behaviour. In fact, omission faults (which are the most restrictive class) and arbitrary faults represent two ends of the fault classification spectrum with other fault types lying in between. Later fault classifications subsume the characteristics of the classes before them. For instance, omission faults can be treated as a special case of either value or timing faults. This spectrum of fault classification is illustrated in Figure 3.1.

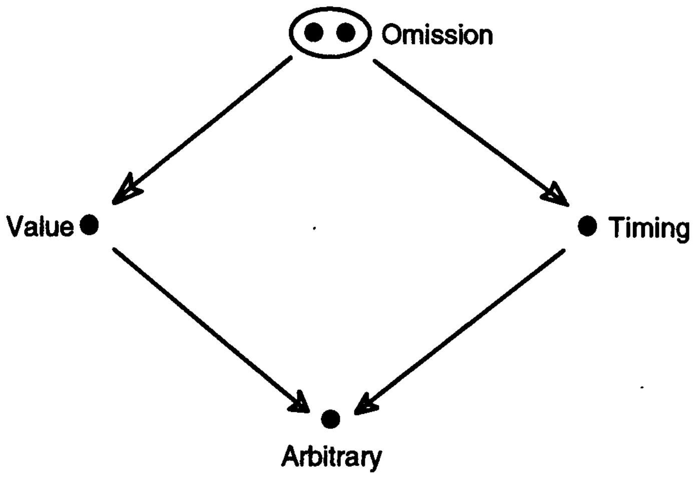
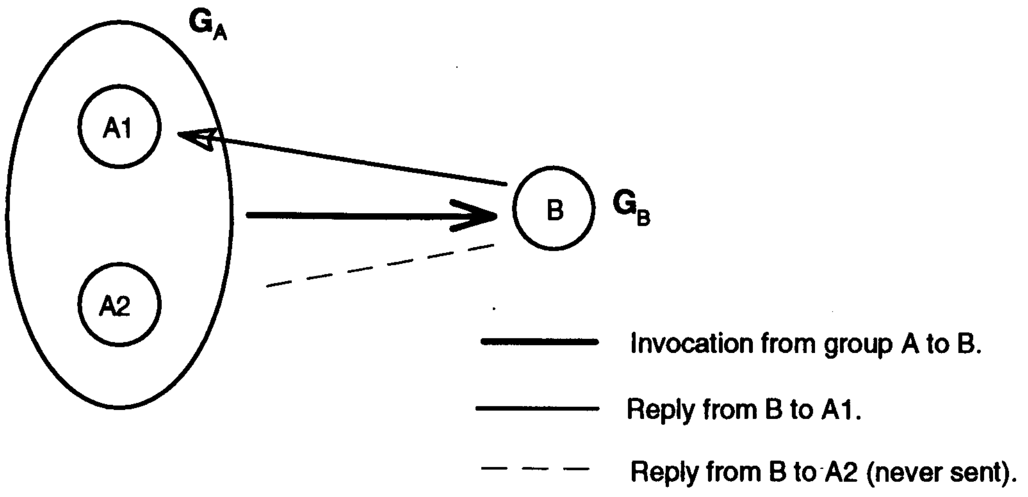


Figure 3.1. Fault and failure classification hierarchy.

The importance of these fault classifications is that the types of fault a component is assumed to exhibit affects the complexity of the replication protocols needed if that component is to be replicated for fault tolerance, availability or performance reasons. This is discussed in more detail below.

3.2. Replication Protocols and Consistency.

The problem of managing replicated objects (or other software components) is more complex than the management of replicated, passive data items. Essentially, the problem of managing replicated objects amounts to that of managing replicated computations and it can best be formulated in terms of the management of object *groups*, where each group will represent a replicated object and groups interact via messages. To ensure consistent behaviour within a group in the presence of concurrent invocations and failures, it is necessary to ensure that incoming invocations and other events such as replica failures or the insertion of new replicas are handled consistently by all of the correct replicas in the group and seen in a consistent order by the users of that group. If appropriate protocols are not used to achieve this, then the states of different replicas can diverge from one another



G_A
A1
A2
B
G_B
Invocation from group A to B.
Reply from B to A1.
Reply from B to A2 (never sent).

Figure 3.2. Object groups and inconsistency.

(see [Pease 80][Frison 82]). For example, consider the situation shown in Figure 3.2. Some object group G_A (replicas A1 and A2 executing in parallel) is invoking an operation on group G_B (a single object B) and B then fails during delivery of its reply to G_A. Suppose that the reply is seen by A1, but not by A2, in which case the subsequent action taken by A1 and A2 can diverge. This problem arises because the failure of B was "seen" by A2, but not by A1.

In the following discussion, both passive and active replication protocols will be considered. Both types of protocol deal with the consistency problems described above, however they do so in different ways and they will often rely upon different underlying communication mechanisms. Different protocols also make different assumptions about the computational model and properties of the applications that they support. In the general case. this makes it impossible to take a simplistic approach to the use of replication and applications must either be designed with particular replication protocols in mind or appropriate protocols must be chosen to match the requirements and features of a given application.

3.3. Passive Replication.

In a passive replica group, only one member of the group (the *primary*) receives, processes and responds to client requests, while the other members of the group act as *backups*, ready to take over the primary's task if it should fail (for example, see [Speirs 89]). To maintain consistency across the replica group, the primary must send checkpoints of its state to its backups. Then, if the backups detect the failure of the primary, they execute an election protocol to select a new primary and the elected replica resumes execution from the most recent checkpoint. This can lead to a break in service when the primary fails, since:

i) it will take a finite, bounded time for the backups to ascertain that the primary has, indeed, failed. The length of this delay will depend upon the failure detection mechanism that is employed.

ii) the execution of the election protocol will introduce a further, bounded, delay.

and

iii) the new primary may have to re-execute operations that were carried out, but not checkpointed by the old primary prior to its failure.

3.3.1. Failure Modes and Failure Detection.

Since requests are only processed by a single replica, passive replication schemes can only handle a restricted class of failures, namely omission failures. Passive replication therefore relies upon processing nodes being *fail-silent.* A fail-silent node is one that exhibits permanent omission failures: that is, when a failure occurs, the node stops functioning. Hence, a fail-silent node never performs erroneous state transitions. Although this is an idealised abstraction of real processors, given sufficient hardware, it is possible to build realistic approximations to such processors (see [Schlichting 83][Ezhilchelvan 90]). Given this fail-silent property and using passive replication, it is then possible to tolerate K failures using K+1 replicas.

To allow other members of the replica group to detect the failure of the primary, passive replication also requires some kind of failure detection mechanism. This can be implemented in several different ways, but possibilities include the use of "I am alive" messages which the primary periodically sends to the other replicas or polling, where the other replicas periodically send a message to the primary to confirm that it is still running. The simplicity of these failure detection strategies is largely due to the assumed fail-silent property of the processing nodes, since the absence of an expected message or reply from a fail-silent node can be taken as a sign that the node has failed.

3.3.2. Communications Support.

Communications support for passive replication can be broken down into two separate problems. First of all, communication within the replica group itself and, secondly, communication between a client and a replicated server. The former will typically be based upon some form of multicast mechanism that will allow the primary to distribute checkpoints to all of its backups, while the latter could be implemented either as a multicast between the client and the server group, in which case the client does not need to know which replica is the current primary, or as a unicast between the client and the current primary, in which case some kind of name service or logical address mapping mechanism must be provided to allow clients to identify primary replicas. Another consequence of using a unicast between clients and replicated servers is that checkpoints must then include information about the requests that the primary has processed, as well as the primary's internal state.

From the point of view of a client, replies from a passively replicated server are simply ordinary reply messages. The replication protocol guarantees that only one replica will respond and, when this response is received, it is guaranteed that the current primary has checkpointed its state to a sufficient number of functioning backups to be able to provide the required level of fault tolerance (e.g. if the replicated service must be able to tolerate K node failures, then K replicas must save the checkpoint as well as the primary). The replication protocols also guarantee that only up to date backups are eligible to become the new primary. This can be achieved by the primary excluding unresponsive backups from the replica group until they have performed update actions, or it may be achieved as part of the election protocol itself.

3.3.3. Repeated Requests.

Another important feature of passive replication protocols is that they must include a mechanism for dealing with repeated requests. Since service is lost for some time after a primary failure, clients may re-try requests. This can lead to problems if the requested operation is not idempotent (i.e. if executing the operation twice is not equivalent to executing it once), so there must be a way for the new primary to recognise a repeated request and respond appropriately. One such method is known as *retained results* [Birman 85], in which a new primary is provided with a record of requests that have already been served and simply re-transmits the results of repeated requests. This also has an impact upon clients, since they must have the ability to re-transmit requests and re-evaluate replies. The latter requires that the client perform backward recovery, returning to the state that it occupied when the failed primary last made a checkpoint. This leads to a trade-off between the frequency of checkpointing and the quantity of state recovery that is required and an appropriate decision must be made on a per application basis.

3.3.4. Passive Replication : Advantages and Disadvantages.

Perhaps the greatest advantage of passive replication is that replicas need not be deterministic in nature. Since only one replica ever performs an operation at any given time, possible non-deterministic behaviour can be hidden by the fact that the group response is provided by the primary and the primary's view of the application's state is imposed upon the backups by checkpoints. Another advantage is that the collation and voting of multiple replies is not required, since the single reply from the primary can be assumed to be correct. On the other hand, the greatest disadvantage of passive replication is the break in service that can occur when the primary fails, since this not only means that clients may have to re-try requests, it can also lead to timing failures if a client is running to a deadline and awaiting a reply from the passive replica group. Also, the fact that passive replication can only tolerate the simplest types of failure represents a further disadvantage.

3.4. Active Replication.

With active replication, all members of a replica group receive and execute all requests and all members of the group send a reply. There is, therefore, no need for checkpointing as with a passive replica group. However, there is a need for new group members to be brought up to date with the existing members before being allowed to receive and respond to requests. Active replication also entails an additional cost in terms of system resources allocated to all members of the replica group so that they can process requests.

Since all members of an active replica group execute at all times, there is no break in service when one of the replicas fails, however some kind of collation mechanism is required so that replies (or requests from a replicated client) can be gathered together correctly. Maintaining replica consistency also becomes more difficult, because there is no primary replica to enforce a consistent view of the current state of the system upon the other group members. To cope with this, special communication protocols are required and replicas must be restricted to performing deterministic computations to ensure that their internal states do not diverge. This is best illustrated by considering a specific model of active replication: the *State Machine.*

3.4.1. The State Machine Model.

The State Machine approach [Schneider 87][Schneider 90] is perhaps the most widely used model of active replication. The state machine method is a general way of implementing fault-tolerant services by replicating individual servers and co-ordinating the interactions of clients with the replicated server group. It has broad applicability in the implementation of distributed and fault-tolerant systems and it provides a framework for understanding replication management protocols for active replica groups.

A state machine consists of *state variables*, which encode its internal state, and *commands* which transform that state and produce output. Each command must be implemented by a deterministic program and the execution of any given command is atomic with respect to the execution of all others. A command may modify state variables, produce some output, or both. To specify the execution of a command, a client sends a message to the appropriate, named state machine, naming the command to be performed and giving any information (parameters, etc.) needed by that command. State machines process requests one at a time, in an order consistent with causality, so clients can be programmed under the assumptions that:

O1 Requests issued by a single client to a given state machine *sm* are processed by *sm* in the order in which they were issued

O2 If the fact that request *r* was made to a state machine *sm* by client *c* could have caused, or potentially caused, a request *r'* to be made to sm by another client *c'* then sm processes *r* before *r'*.

The outputs from processing a request may be sent to some kind of actuator, to some other peripheral device or to state machine clients that are awaiting responses.

From the properties described above, it can be seen that the defining characteristic of a state machine is that it specifies a deterministic computation that reads a stream of requests and processes each one, producing any necessary outputs. Semantically, each output from a state machine is completely determined by its initial state and the sequence of requests that it processes, independent of time or any other activity in the system. The use of active replication to construct a fault tolerant state machine out of individual state machine replicas is, therefore, relatively simple: so long as each replica running on a non-faulty processor starts in the same initial state and processes the same requests in an identical order, it can be guaranteed that the internal states of the non-faulty replicas will remain consistent and not diverge. The requirement that replicas execute the same requests in an identical order can be decomposed into two separate, specific constraints on the dissemination of information to the replica group.

Agreement

Every non-faulty state machine replica receives all requests.

Order

Every non-faulty state machine replica processes the requests it receives in the same relative order.

These agreement and order properties can be implemented in a number of different ways, however the most commonly used protocols are based upon the use of an atomic broadcast mechanism.

3.4.1.1. Communication Support for State Machines : The Atomic Broadcast.

An atomic broadcast is a multicast communication mechanism that possesses the following three basic properties :

Termination

If the initiator of the broadcast (the sender) does not fail, then the broadcast will complete successfully within a bounded time. That is to say, if the sender begins the broadcast at some time t on its local clock, then the broadcast will be guaranteed to complete by some fixed time $t+\Delta$, where Δ is a known and bounded quantity.

Atomicity

If the initiator of the broadcast does not fail, then all the non-faulty, intended recipients of the broadcast will receive the message. Alternatively, if the sender fails during the broadcast, then *either* all of the recipients will receive the message *or* none of them will. (No partial broadcasts.)

Order

All broadcasts are received in the same order at all replicas.

The use of an atomic broadcast protocol for communication with a replicated state machine guarantees the State Machine Agreement property and the state machine Order property can be guaranteed by having all replicas process requests in the order in which they arrive, (since the arrival of requests is ordered by the atomic broadcast). The use of an atomic broadcast in this way will also preserve the causal relationships between requests as required by the state machine causality assumptions, ***O1*** and ***O2***.

Although atomic broadcast mechanisms have been included in a number of existing systems, the detailed implementation of an atomic broadcast primitive depends upon the types of failures that are to be tolerated and the underlying low-level communications support upon which the atomic broadcast is based. However, in general, the participants in an atomic broadcast will have to exchange and compare messages that they have received to ensure that the termination, atomicity and order properties are met. This may involve one or more rounds of message exchange between all participants and the cost of executing an atomic broadcast can be high in terms of both processing requirements and communications overheads. This topic is discussed further in chapter six, where the implementation of two different atomic broadcast protocols is described (one capable of tolerating arbitrary failures, the other of tolerating omission failures).

3.4.2. Failure Modes and Failure Detection.

Where passive replication techniques are limited to handling permanent omission failures and rely upon fail-silent processing nodes, active replication schemes can be used to tolerate any class of failure, including arbitrary (Byzantine) failures. The number of replicas that are required and the precise details of the replication protocols involved then depend upon the type of fault that is to be handled. At one extreme, if it can be assumed that processing nodes are fail-silent, then active replication can tolerate K faults using only K+1 replicas. All non-faulty replicas reply to all requests and a client of such an active replica group can simply accept the first response that it receives (since the fail-silence property ensures that it will be correct). The fact that all non-faulty replicas reply to all requests also means that there will be no discernible break in service when an individual replica fails. However, an atomic broadcast or similar protocol must be used for communication with the replica group to ensure that messages will be received in a consistent order by different replicas. The cost of executing such a protocol must, therefore, be included in the time taken to communicate with the group.

At the other extreme, tolerating K Byzantine faults requires either 2K+1 or 3K+1 replicas: the former if some kind of signature mechanism is available to allow messages to be authenticated (e.g. a mechanism based upon [Rivest 78]) and the latter if an authentication mechanism is not available. In either case, more complex agreement protocols are required for communication with the group, since any protocol that is used needs to be able to tolerate arbitrary faults in one or more participants. The replica group must also be able to form a majority decision regarding the actions it is to take and the outputs that are to be produced, so some kind of majority voting mechanism is usually required. As in the fail-silent case, a client of the active replica group will receive replies from all of the non-faulty replicas, but some, or all, of the faulty replicas may also produce replies. This means that the client also needs to perform message collation and majority voting in order to determine the correct reply for any given request. Once again, there is no discernible break in service when a failure occurs, but the overhead on interactions with the replica group will include the cost of message collation and majority voting as well as the agreement protocol.

3.4.3. Active Replication : Advantages and Disadvantages.

Considering disadvantages first, there are three major problems with active replication. Firstly, there is the cost in system resources (processing time, etc.) of having all replicas execute all requests. Secondly, there is the cost of the replication protocols themselves. As mentioned above, atomic broadcasts and other agreement protocols can be very complex and their execution can be costly in terms of both processing time and communication overheads. Further overheads are introduced by the need for message collation and, in non-fail-silent replica groups, majority voting amongst replicas. This, in turn, places extra demands on the available resources within a system and response times for active replica groups are typically slightly degraded when compared with corresponding, unreplicated services. The third major disadvantage of active replication is that replicas must be deterministic in their execution, otherwise the internal state of different replicas may diverge. (Note that the State Machine model is particularly well-suited for active replication precisely because State Machines are restricted to deterministic programs.)

In spite of such disadvantages, active replication schemes have been used in several systems. This is largely due to two major advantages of the active approach. First of all, response times for an active replica group are not degraded when failures occur, so active replication can be particularly suitable for applications that have high reliability requirements or include timing constraints. This issue is discussed further in section 3.5. The second major advantage of active replication is that it can tolerate arbitrary classes of fault. Active replication can therefore be used in a wider range of applications than other replication strategies and, in situations where it cannot be assumed that processors or processing nodes are fail-silent, active replication is, in fact, the *only* replication strategy that can be used to provide fault tolerance.

3.5. Replication in Real-Time Systems.

If replication is to be used to provide fault tolerance for a real-time application, the impact of the replication protocols themselves on the overall performance of the system must be taken into account. A simple first analysis might suggest that passive replication is more appropriate for the real-time applications domain, since complex agreement and ordering protocols do not need to be executed when service requests are made. The fact that passive replicas can be permitted to perform non-deterministic computations can represent another advantage, since this would seem to allow greater flexibility of execution and make it easier to cope with events and signals coming from the system's environment. However, from a real-time perspective, passive replication does suffer from disadvantages. There are processing and communications overheads for constructing checkpoints and transmitting them from the primary to the rest of the replica group. There is also the problem that passive replica groups suffer a break in service whenever the primary replica fails (or is detected to have failed by the backups). When this happens, the length of time for which the service is unavailable is determined by:

$$L_{fd} + T_{elect} + T_{update}$$

where L_{fd} = Latency of the replica group failure detection mechanism,

T_{elect} = Time to elect new primary replica (i.e. time to run election protocol),

and T_{update} = Time for new primary to bring itself up to date from the most recent checkpoint.

This break in service can lead to problems elsewhere in the system, particularly when other tasks miss their deadlines while awaiting replies from the passive replica group. In the worst case, there is also the risk that a primary replica might fail when the system is already operating under emergency or heavy-load conditions. If this was to occur, the temporary loss of service from the passive replica group might cause a more serious failure of the system as a whole. Such a situation would be unacceptable in many real-time applications and passive replication may, therefore, be regarded as being unsuitable for real-time services. The fact that passive replication assumes fail-silent hardware can pose further problems, since there are some applications (e.g. in safety-critical systems) where fail-silence cannot be assumed unless appropriate hardware redundancy is employed to support it, but where hardware implementation of fail-silent nodes may, nevertheless, be infeasible or inappropriate.

Active replication, on the other hand, does not suffer from these disadvantages. Since all replicas execute all requests, the failure of an individual replica will go unnoticed from the point of view of the replica group's clients (i.e. no loss of service). Furthermore, active replication is not limited to fail-silent hardware. For these reasons, active replication would appear to be the better replication strategy for real-time systems and for applications having high dependability requirements. However, the disadvantages of active replication must also be taken into account. As described earlier, there is the cost in terms of system resources of having all replicas executing in parallel at all times. Fortunately, this does not necessarily present any major difficulties and it can largely be regarded as a resource allocation problem. More important is the cost of executing the necessary agreement and replica consistency protocols. In a real-time environment, the trade-off between increased fault tolerance and degraded response times must be weighed very carefully and situations may arise where a system does not have sufficient processing or communications capacity to handle the extra overheads of communication between active replica groups. In such cases, an alternative fault tolerance strategy would have to be adopted. Finally, there are the problems associated with non-determinism. The restriction of active replicas to deterministic computations is particularly awkward for real-time systems because, as we shall see in the following section, the potential for non-determinism in real-time programs can be high.

3.6. Sources of Non-determinism.

Although agreement protocols can ensure that the internal states of different replicas do not diverge as a result of processing incoming messages in a different order, state divergence can occur in other ways. Even if the restriction is made that all programming language constructs in the programs performed by active replicas must be deterministic, there are several other possible sources of non-deterministic behaviour. For example, consider the situation where replicas each set a timeout while waiting for some message to arrive. As a result of transmission delays in the network connecting the replicas, it is possible that some replicas may see a copy of the message before their timeout expires, while others do not. If this occurs, then the states of those replicas that saw the message in time will, in the general case, diverge from the state of those replicas that didn't see the message in time. A similar situation may arise with any other real-time constraint, where some replicas meet their timing constraint while others, due to varying message or processing delays, do not. In both of these cases, the fact that all replicas were executing deterministic programming language constructs and that all invocations were agreed and ordered does not have any bearing on the potentially divergent behaviour of the application. The problem here is that the replicas were each responding to an entirely local, internal event (in the above examples, the expiry of some timer) and such internal events must also be agreed and ordered across the replica group.

Taking other possible sources of non-determinism into account and considering typical real-time and distributed applications, six general classes of activity can be identified that may lead to state divergence:

1. *Non-deterministic language constructs.*

   The execution of non-deterministic language constructs by the replicas themselves has already been mentioned as a source of state divergence. The usual (and, typically, easiest) solution to this problem is to adopt an appropriate replication model (such as the state machine) that does not allow replicas to execute such non-deterministic primitives.

2. *Invocation and group messages.*

   Non-determinism due to replicas serving different requests in different orders or observing group membership changes at different times has also already been discussed. Again, the solution is to adopt an appropriate replication model such as the state machine and to execute appropriate agreement and order protocols to ensure that replicas process the same messages in the same order.

3. *Environmental interactions.*

   If replicas are allowed to interact directly with their environment, for instance receiving sensor readings or alarm signals, then it is possible that different replicas could receive different values or receive signals at different times. Such interactions with an external environment may cause different replicas to follow different execution paths and their states to diverge.

4. *Event handling and mode changes.*

   In a sense, this may be regarded as a special case of an environmental interaction. If replicas may operate in different modes, depending upon recent events in the system's environment or the current state of the application itself, then changes from one mode of operation to another must be made consistent across the replica group. The classic case of this type of situation arises when replicas are waiting for two or more possible events and subsequent processing depends upon which event has occurred. If different replicas detect different events, then they may perform totally different operations, to all intents and purposes immediately

5. *Timing constraints.*

   As in the situations described earlier, the use of timeouts or timing constraints can lead to state divergence if some replicas regard the timeout or timing constraint to have expired while others regard it as having been met. Such a discrepancy in the replica group view may arise as a result of clock synchronisation differences, variance in message delays between replicas or from varying processing delays.

6. *Internal concurrency.*

   The presence of internal parallelism in replicas, for instance the replicas of a multi-threaded object, can cause state divergence if parallel threads are allowed to interact. This could either be due to uncontrolled, concurrent threads corrupting internal state information in some replicas or, if some form of concurrency control is being employed, threads in different replicas interacting at different points in their execution or different replicas enforcing different concurrency control decisions.

Unfortunately, classes three, four and five represent types of activity that are common in a wide range of real-time applications and the use of internal concurrency may also be common, either for reasons of performance or as a result of a system's environment including a high degree of parallel activity. It may therefore appear that the use of active replication to support fault tolerance in real-time services is also doomed to failure as a result of replica state divergence. However, this is not necessarily the case.

3.7. Solving the Non-Determinism Problem.

By and large, if a system has enough resources to support it and if the problems associated with non-deterministic computations can be solved, the use of active replication can be an effective fault tolerance technique for real-time applications. The resource allocation issue, both for the replicas themselves and for the underlying support mechanisms that are required, can only be addressed on a case by case basis, since different applications will have different requirements. The "non-determinism problem", on the other hand, does admit more general solutions and the remainder of this section describes some possible approaches: first, straightforward prevention of non-determinism as enforced in the MARS and SIFT systems; second, some less restrictive approaches that have been used in the Delta-4 system and, finally, some proposals for an application level solution to this problem.

3.7.1. Prevention of Non-determinism.

The MARS system described in the previous chapter and the SIFT system [Melliar-Smith 82] [Weinstock 80] both make use of active replication to provide fault tolerance and both systems follow the state machine model in restricting all tasks or processes to purely deterministic computations. For example, in SIFT all processes are iterative, repeatedly accepting input data, performing some specified, deterministic calculation and producing an appropriate output. Given this determinism property, the only possible source of state divergence in either system is then the ordering of requests or messages at different replicas.

To prevent state divergence arising from this source, both systems rely upon a combination of static scheduling and broadcast communication mechanisms. In MARS, all communication between tasks is by means of state messages and all access to the underlying communications medium is pre-scheduled during system design, while in SIFT, hardware support is provided for broadcast messages and an interactive consistency algorithm is periodically executed by all the replicas of a process to agree upon the set of messages that have arrived or are about to arrive. Hence, in both systems, it can be ensured that all replicas process the same set of messages in the same order and thus remain in mutually consistent states.

This restrictive approach to the problem of non-deterministic computations in active replica groups is effective and has been adopted in a number of systems, however it does tend to constrain the run-time behaviour of a system. For example, non-deterministic language constructs cannot be used (including useful constructs such as the `select...accept` and rendezvous mechanisms used in Ada tasking) and internal parallelism within individual tasks cannot easily be supported. Problems can also arise if tasks are allowed to be pre-empted at arbitrary points in their execution, so pre-emptive scheduling disciplines must usually be avoided. (It is notable in this regard that both of the systems mentioned above use a static scheduling policy in which task start and end times are pre-defined.)

3.7.2. Non-determinism in the Delta-4 System.

In the Delta-4 system, two different solutions to the problem of non-determinism in replica groups have been adopted: one based upon the use of active replication and the other based upon the use of an alternative replication strategy known as *semi-active* or *leader-follower* replication. In both cases, the approach that is taken to non-deterministic computations is less restrictive than that in the MARS or SIFT systems, however it can still be ensured that the internal states of the members of a replica group will remain mutually consistent and not diverge.

3.7.2.1. Active Replication for Multi-threaded Objects.

In the case of active replication (see [Chereque 92]), replicas must still conform to the state machine model, however certain non-deterministic mechanisms are allowed, notably pre-emptive multi-threading within objects. As in the MARS and SIFT systems, non-determinism due to the ordering of requests or messages at different replicas is prevented by using an appropriate broadcast communications mechanism to ensure that replicas receive the same messages in the same order. Internal non-determinism due to multi-threading within objects is then handled at the task scheduling level by ensuring that thread scheduling occurs at the same place in computation and uses the same scheduling data in all replicas. This is achieved by having the thread scheduler invoked exclusively from well-defined points in computation, such as at the execution of certain well-defined language constructs (e.g. rendezvous, `select`, etc.) or on entry to or exit from compilation units, system or standard procedures or well-chosen library procedures. In essence, this can be viewed as a co-routine based implementation of multi-threaded objects.

Once again, this can be an effective strategy to adopt, but it does have certain drawbacks. First and foremost, it requires alterations to be made within the task scheduling primitives of the real-time operating system upon which an application is to run. While this may be feasible for applications that are being developed upon a bare hardware platform (since such an application will incorporate its own task scheduler), it will be impossible for applications that are being developed to run on an existing operating system, unless the existing scheduler already provides the necessary support or it can be modified to do so. Another disadvantage with the scheduler based approach is that it can only really be applied within a fixed, static scheduling regime, since it depends upon all replicas being de-scheduled at the same, fixed point during their execution and this may not necessarily be true in a dynamic system.

3.7.2.2. Semi-active Replication.

The semi-active replication strategy used in the Delta-4 XPA (Extra Performance Architecture) system [Barrett 90] represents another approach to the non-determinism problem. Semi-active, or leader-follower, replication is an attempt to combine the advantages of passive and active replication techniques. In a semi-active replica group, all replicas receive and process all requests, but only a single replica, the *leader*, will send a reply. The leader is also responsible for dictating the order in which service requests should be processed by the replica group and for informing the other members of the replica group (the *followers*)[1] of the outcome of any non-deterministic choices made during the course of execution. This is done by means of *synchronisation messages*. When it selects a new request to be processed, the leader sends a synchronisation message[2] informing the followers of the request that has been chosen. Similarly, when the leader makes a non-deterministic choice, it sends a synchronisation message informing the followers of the outcome of that choice. Hence, the followers will execute the same sequence of operations as the leader and the leader's choice can be forced upon the followers when they are faced with a non-deterministic decision. Problems with state divergence are, therefore, avoided.

As well as handling the ordering of requests and the results of non-deterministic program constructs, the synchronisation message mechanism can also be used to allow the pre-emption of requests. In those cases where the processing of a request may be pre-empted by

[1]Hence the alternative name, leader-follower replication.
[2]In fact, synchronisation messages are sent on behalf of the leader by a task in the underlying communication layer of the leader's host node (or its network interface). This is to relieve the leader of the responsibility for managing message transmission. However, but for performance implications, this is equivalent to the leader sending the synchronisation messages itself.

the arrival of a higher priority message or signal, the pre-emption must be synchronised across all replicas to ensure replica determinism. In a semi-active replica group, this can be achieved by introducing the concept of a *pre-emption point*, which is a pre-defined point in a software component at which it may be pre-empted. Each time the leader reaches a pre-emption point, a counter is incremented. When a message arrives at the leader, a check is made to determine whether this message requires the leader to be pre-empted. If this is so, the pre-emption point at which this will take place is selected (given by the current value of the counter plus one) and a synchronisation message is sent to the followers containing this value and identifying the message that caused the pre-emption. On arriving at the assigned pre-emption point (i.e. when their counters match the assigned value), each replica then begins to process the pre-emption.

The use of semi-active replication to tackle the non-determinism problem has two particular advantages. Firstly, since the ordering of requests across the replica group is enforced by the synchronisation message mechanism, communication with a semi-active replica group only requires a reliable (atomic, *unordered* as opposed to ordered) multicast protocol (see chapter 6). Secondly, the ability to pre-empt the processing of a request can be extremely useful in real-time applications. However, the use of semi-active replication also has some disadvantages. Like passive replication, semi-active replication requires fail-silent processing nodes, since only a single replica (the leader) replies to requests. Further, for the pre-emption point mechanism to work, the followers must always be executing at least one step behind the leader, where a step constitutes the receipt of a synchronisation message due either to a pre-emption or to the consumption of an input message by the leader. This could lead to followers falling too far behind the leader, although "dummy" synchronisation messages (which would also double as "I am alive" messages) could be sent periodically by the leader to avoid this problem. Finally, it is unlikely that the pre-emption point mechanism could be made completely transparent to the application programmer, since it involves the insertion of appropriate pre-emption point code into the application software.

3.7.3. An Application-level Approach.

The final approach to the non-determinism problem, proposed in this thesis, is to attempt to prevent state divergence at the application programming level, as and when it occurs. This objective will be achieved by developing a real-time object model, based upon the state machine, which imposes certain restrictions upon the computational model used for applications, but which also copes with the six different classes of non-deterministic activity listed earlier. The general approach in each of the six cases is *either* to impose a particular structuring of activity at the application level such that state divergence can be prevented, *or* to develop equivalent, message based implementations of non-deterministic constructs such that agreement and ordering protocols can be performed where required.

The details of the real-time object model will be discussed more fully in the following chapter, however the strategies employed to handle non-deterministic activity are described below.

3.7.3.1. Non-deterministic Constructs and Incoming Messages.

These are handled in the usual State Machine fashion, by restricting replicas to deterministic computations (except where the constructs to be described below are involved) and executing appropriate agreement and order protocols on invocations between replica groups.

3.7.3.2. Environmental Interactions.

For environmental interactions such as sensor readings or alarm signals, the easiest approach is to convert such stimuli into message based primitives. The message associated with a particular reading or signal can then be agreed and ordered as for any other communication with the replica group. For signals from intelligent device controllers, mapping the signal to some kind of message may be relatively straightforward, while dumb devices would require an appropriate interface module or interface object to be implemented in order to carry out the required marshalling of values and transmission of the message. However, in principle, the technique remains simple.

3.7.3.3. Event Handling and Mode Changes.

In this case, some mechanism is required to limit the set of invocations that an object is prepared to accept at a given time. One possible solution is to have an active object (i.e. one containing an independent, internal thread of control) which executes a **select...accept** mechanism similar to that used in the Ada language. By mapping possible events to invocations and, at any given time, only accepting those invocations that correspond to meaningful events, the future execution path of the object can be restricted to an appropriate subset of its possible modes of operation. The correspondence between events and subsequent operational modes is also easily supported, since an **accept** operation may have associated with it a defined sequence of operations that are to be executed if the **accept** succeeds. This seemingly non-deterministic mechanism can be used to handle events in this way because it is possible to implement the **select...accept** construct in such a way that replica consistency is preserved, even though the selection of messages remains non-deterministic at the application level. This is achieved as follows:

Consider the situation where a task is prepared to accept one of a number of messages, m_1,m_2...m_n, with a timeout of t seconds should no message arrive. This is shown, in terms of the **select** construct, in figure 3.3.

```
select
    accept m1 → action1 ;
  □
    accept m2 → action2 ;
  □
    ...
  □
    accept mn → actionn ;
  □
    delay(t) → timeout_action ;
end select ;
```

Figure 3.3. **select...accept** *language construct.*

One possible implementation of this construct, at the level of processes and messages, is shown in figure 3.4. The messages $m_1 \ldots m_n$ are identified by the port upon which they arrive ($P_1 \ldots P_n$ respectively) and all messages for the process S are placed in a message queue, from which they are retrieved using the `receivefrom(P,t)` primitive. Now, if an atomic broadcast protocol, or similar mechanism, is used to ensure that all the replicas of S have identically ordered message queues and if the `receivefrom` primitive returns the first appropriate message found in the queue, then it can be guaranteed that in those cases where at least one message arrives in time, each replica will process the same message.

```
process S
   var m : message,
       t : timevalue,
       P : setof ports

       P := P[Pi...Pn]
       t := <timeout value>
       m := receivefrom(P,t)
       if m ≠ null → if  m.port=P1 → action1     /* m = m1 */
                     □   m.port=P2 → action2     /* m = m2 */
                              ...
                     □   m.port=Pn → actionn     /* m = mn */
                     fi
       □ m = null → timeout_action
       fi
end S
```

Figure 3.4. Implementation of **select** *construct.*

However, since the input operation is non-blocking and times out after some interval, t, different replicas may still perform different actions. This can happen if an appropriate message is placed in the message queues of some of the replicas just before the timeout expires, while other replicas do not see the message in time. To solve this problem, what is needed is some means whereby the timeout event can be agreed and ordered just like an incoming message. This can be achieved by having each replica send a *marker* message to itself when it detects the expiry of the timeout (see figure 3.5). This self-directed message will be broadcast to the replica group and agreed and ordered in the same way as any other. It can therefore be guaranteed that *either* all replicas will select the same message from a member of P *or* all replicas will select the marker message. Replica consistency will thus be maintained.[3] Furthermore, since message queues are ordered identically at all replicas, prioritised messages can also be handled consistently by having each replica await the arrival of the marker message and then search through its message queue, as far as the marker, for the highest priority acceptable message that has arrived. Clearly, if some high priority message, m_H, is selected by one correct replica, the same message will be in the queues of the other correct replicas and will, similarly, be selected.

```
procedure receivefrom(P : setof ports, t : timevalue)
    returns m : message
begin
    within t do
        { m := RECEIVEFROM(P) }    /* Returns first message  */
    timeout:                       /* in queue from Pi∈ P.    */
        { send(self, marker)
          m := RECEIVEFROM([P,self]) }
    od
    if m = marker → m := null
    □ m ≠ marker → skip
    fi
end
```

Figure 3.5. The generic input function.

The implementation of the `receivefrom()` primitive shown in figure 3.5 has been referred to as the *generic input function* since it can be used as the basis for a deterministic implementation of any type of message input primitive, not just the **select...accept** construct (for further details, see [Tully 90][Tully 91][Shrivastava 92b]).

3.7.3.4. Timing Constraints.

To handle timing constraints, it is possible to develop a message based implementation of the "temporal scope" construct that relies upon using the deterministic implementation of the **select...accept** mechanism described above. Since the violation of a timing constraint will then be represented by a message, it can be agreed and ordered for the entire replica group.

For example, consider the following timing constraint :

[3] Note also that it must be assumed that the atomic broadcast used to distribute the marker message includes some form of collation and duplicate removal mechanism to ensure that each replica will only receive a single copy of any given marker.

```
do
  S ;
start_in(EST,LST) ;   [ Early_start : A ; Late_start : B ] ;
finish_in(EFT,LFT) ;  [ Early_finish : X ; Late_finish : Y ] ;
```

The interpretation of this constraint is that operation S should be executed, starting within the closed interval [EST, LST] and completing within the closed interval [EFT, LFT]. If S is started prior to EST, the exception Early_start will be raised and exception handler A will be executed. Similarly, if S has not started by LST, the exception Late_start is raised and handler B is executed. Completion before EFT is signalled by Early_finish and handled by handler X, while late completion (i.e. after LFT) is signalled by Late_finish and handled by handler Y. This construct permits an equivalent implementation of the form :

```
par
  send(self,"start") ; S ; send(self,"finish") ;
||
  select
    accept "start" → A ;
  □
    delay(EST) →
        select
          accept "start" →
                    select
                      accept "finish" → X ;
                    □
                      delay(EFT) →
                          select
                              accept "finish" → null ;
                          □
                              delay(LFT) → Y ;
                          end select ;
                    end select ;
        □
          delay(LST) → B ;
        end select ;
  end select ;
end par ;
```

where the statement "**delay**(t) → S ;" represents a guarded command that will delay until time t and then execute S.

The way in which this alternative implementation works is quite simple. Two parallel threads are created, one of which sends a "start" message, executes operation S and sends a "finish" message[4], while the other thread waits for the incoming messages using a nested

[4] Note that, for absolute correctness, the transmission of the "start" message and the beginning of operation S must be atomic. Similarly, the end of operation S and the transmission of the "finish" message must also

select mechanism. Each **select** has only two branches: either **accept** an incoming message (`"start"` or `"finish"`) or, after the pre-defined delay, proceed to the next level of nesting. Now, consider the situation when execution begins. If the first **accept** for message `"start"` succeeds, then the parallel thread has begun execution too early (i.e. before `EST`), so operation `A` (the exception handler operation) is performed. On the other hand, if execution of `S` is not begun too early, the first delay statement (for `EST`) will expire and the next level of nesting will be entered. Here, if the `"start"` message arrives before the next delay expires (`LST`), then all is well and execution proceeds. However, if **delay**`(LST)` expires, then the parallel thread has not begun execution on time and the handler operation `B` is performed. This pattern of execution is repeated for the earliest and latest finish times (`EFT` and `LFT`). It can be seen that the net effect of this mechanism is the same as the net effect of the timing constraint described earlier, however in this case, exceptional events such as `Late_start` or `Early_finish` are mapped to the arrival of start or finish messages and an appropriate agreement protocol can therefore be executed for these events.

3.7.3.5. Internal Concurrency.

In this case, the best approach is to impose a restriction upon the computational model such that state divergence cannot occur. First of all, an operation is only allowed to generate internal threads using the **par...end par** construct if the threads thus created do not interact with one another, other than via the exchange of messages or via invocations to common objects. This ensures that all such interactions are visible at the message level and can, as required, be agreed and ordered between replicas. The second restriction is that active objects may only contain multiple threads if those threads meet the same criteria: i.e. that they do not interact other than via the exchange of messages of via invocations to common objects. This even applies when an internal thread needs to access an object's internal state information: such access must be via an operation invocation, although in this case, the invoked operation may be private rather than public. Under these conditions, it can

be atomic. A correct implementation of this mechanism must, therefore, ensure that the thread executing S cannot be suspended immediately after transmitting "start", or immediately before transmitting "finish".

be ensured that the thread's activity is agreed and ordered both at remote objects and locally to the active object itself.

3.7.4. Impact of the Proposed Approach.

If the mechanisms proposed in section 3.7.3 are to be used, some thought must be given to the impact that they will have on the design of systems and the performance of such systems at run-time. From the point of view of system design, the restrictions that have been imposed are unlikely to prove too limiting, since they are mainly recommendations of the way in which particular activities should be supported, rather than prohibitions. For example, internal parallelism in objects is not actually disallowed, it is simply restricted to certain well-defined mechanisms.

Considering system performance, the mechanisms that will have the greatest impact are those that have been proposed for event handling and timing constraints. The use of the `select...accept` construct in the implementation of the temporal scope primitives and the use of marker messages in the deterministic implementation of select will introduce execution overheads at run-time. These overheads will vary from system to system, being dependent upon factors such as the latency of the underlying communications protocols that are used. Chapter 7 considers this subject in greater detail and gives a brief analysis of the run-time costs associated with the suggested timing constraint mechanism.

3.8. Chapter Summary.

In this chapter, the use of replication to provide fault tolerant, abstract software components has been examined. A brief description of the classes of fault and failure which can be attributed to components in a distributed system was given, before moving on to a general description of both passive and active replication techniques. The major features of passive replication protocols were discussed, including the need for fail-silent host computers and the freedom of passive replicas to execute non-deterministic computations. Active replication was then considered in the form of the state machine model, illustrating the ability of active replica groups to tolerate arbitrary failures, but also the requirement for deterministic execution in active replica groups. The relative advantages and disadvantages of these two types of replication strategy were then compared in terms of their usefulness in a real-time environment: the conclusion being that active replication, in spite of its requirement for determinism, was the most appropriate technique. Finally, some of the major sources of non-deterministic behaviour (and, hence, replica state divergence) were identified and mechanisms proposed that can allow these types of activity to be undertaken in an active replica group without compromising the consistency of replica states.

Chapter 4.

A Model for Fault-Tolerant Real-Time Objects.

This chapter presents a model for real-time objects that is based upon an extended version of the state machine (see also [Shrivastava 91a]). The model includes a programmed error recovery mechanism that is based upon a combination of atomic transactions and exception handling. Active (threaded) objects are also supported and language constructs are provided that allow real-time constraints to be expressed and enforced at the application programming level. The notion of time can therefore be regarded as a first-class programming entity, similar to any other block-structured programming language primitive.

The final important feature of the object model proposed here is that it allows certain, controlled forms of non-deterministic behaviour, while still being amenable to active replication. This may be achieved by adopting the computational restrictions and implementation techniques described at the end of the previous chapter.

4.1. Object Structure.

The object oriented programming model provides a good structuring technique for large or complex software systems. The data abstraction and encapsulation features of the object oriented approach can be used to develop a well-defined decomposition of a large system into an appropriate set of functional sub-units with a known pattern of information flow between them.

An object is defined to be an instance of some type or class and it consists of a number of *instance variables* which define its internal state and a set of operations or *methods* which serve to define its externally visible behaviour. All interactions between objects take place by means of method invocations and the only way to gain access to an object's state information is by invoking one of its methods. Objects therefore provide a natural boundary for fault containment and the restriction of object interactions to method invocations across defined interfaces aids traceability and provides some support for damage assessment.

4.1.1. Object Granularity.

One of the issues that arises in the context of object-oriented systems is the *granularity* of objects. In a pure object-oriented system such as Smalltalk, all entities are objects, even down to the level of individual integers or characters. All interactions are method invocations (in the case of Smalltalk, viewed as message passing) and inheritance is used extensively to develop more sophisticated objects from existing ones. At the other extreme, objects in some systems are very large. For example, in CLOUDS [Dasgupta 91], objects may encapsulate entire programs and span several nodes in a distributed system.

For reasons of performance and manageability, choosing an appropriate object granularity can be an important design decision and this is particularly true for real-time applications. If a very small granularity is used, it may entail considerable overheads for inter-object communication during normal processing and, in a time-constrained environment, this would usually be unacceptable. On the other hand, a large granularity does not offer the usual structuring advantages in terms of data abstraction, encapsulation and the management of complexity within programs. Fortunately, in many cases, the appropriate granularity for objects in a given real-time application is dictated by the application itself. For example, by having objects represent individual devices or control sub-systems, a natural separation of concerns can be achieved at the software level, corresponding to the structure of the application and the underlying system hardware. This level of granularity will also usually provide an acceptable level of abstraction without incurring excessive overheads at run-time and the object model presented here is geared towards such medium grain entities. This view is similar to that taken in the development of the Alpha kernel [Northcutt 87] and the reasoning upon which it is based follows similar lines.

4.1.2. Active Objects.

It is also often advantageous for real-time objects to be *active*. An active object is one which contains its own internal process or thread of control and which is capable of carrying out processing independently of invocations received from other objects. The degree of autonomy that this provides allows continuous monitoring and control activity to be supported within the object-based framework without having to rely on any kind of explicit process-based mechanism. For most applications, a combination of active and passive objects will usually be appropriate, with the passive sub-systems of the application providing required services to the active processing sub-systems.

4.1.3. Inter-Object Communications.

For interactions between objects, both synchronous and asynchronous invocation mechanisms should be available. The former can then be used to support the traditional client-server style of programming while the latter may be useful for supporting activity in event-driven systems where certain conditions can cause invocations to be generated for which replies are not required. Since it is the *abstraction* of synchronous and asynchronous invocations that is important rather than the actual implementation, this could be achieved by having separate synchronous and asynchronous communication mechanisms or by implementing both types of invocation primitive on a single synchronous or asynchronous communications layer.

4.1.4. Atomic Actions and Exceptions.

Given that the interfaces between objects provide a logical boundary for fault containment, fault tolerance mechanisms can be introduced to ensure that erroneous state information and incorrect results cannot be propagated freely across invocations. Other mechanisms can then be used to provide appropriate error detection and recovery within the objects themselves. One combination of fault tolerance techniques that is particularly well-suited to this task is the use of atomic actions alongside a suitable exception handling construct. By executing all operations as atomic actions, an effective error recovery mechanism is provided within objects and interference between concurrent invocations is prevented. The use of atomic actions in this way can also help in allowing objects to be regarded as a form of state machine, since state machine operations must also execute atomically (with respect to concurrency). Exceptions can then be used to signal failed invocations to the caller, where exception handlers can provide further recovery if it is required.

4.1.5. Initial Overview.

In the following pages, several features of this real-time object model, including the atomic action and exception handling mechanisms will be considered in greater detail, however it will be worthwhile at this point to summarise the model's basic structure. Thus far, five basic properties have been identified:

1. Objects are of medium granularity, typically corresponding to specific devices or control sub-systems.

2. Objects may be active, containing one or more threads of control. (Typically, the number of threads required in a multi-threaded object will be small, but no explicit limit is placed upon this.)

3. Objects may communicate via synchronous or asynchronous invocations.

4. An object is only permitted to have a single interface.

5. All operations are performed as atomic actions.

The restriction that objects are only allowed to possess a single interface is enforced to maintain a degree of simplicity in the interactions between objects. In principle, there is no fundamental reason why this restriction should not be relaxed to allow an object to possess multiple interfaces and to support some form of interface trading or interface checking mechanism, however a number of test applications have already been considered within the framework of this model and it is interesting to note that the need to support objects with multiple interfaces has not yet been apparent.

Figures 4.1 and 4.2 illustrate some of the general features of a programming notation based upon this object model. The syntax for general language constructs is broadly similar to Pascal or Modula-2, however other features, to be described later, have been based upon other languages or systems.

```
INTERFACE Integer_Stack IS

   Push( IN integer ) returns boolean ;
   Pop returns integer ;
   Clear_All ;

END.                          /* End of interface declaration. */
```

Figure 4.1. General language features : Interface Declaration.

```
OBJECT Integer_Stack IS

   /* Implements a stack for up to 100 positive integers. */

STATE                               /* Private state variables. */
   integer top, Stack[100] ;

METHOD Push (IN integer i) returns boolean
action
   if top < 100 then
      begin                      /* If stack not already full,  */
         Stack[top] := i ;       /* add element to top of stack */
         top := top+1 ;          /* and return TRUE.            */
         return TRUE ;
      end
   else
      return FALSE ;             /* Return FALSE if stack full. */
end action ;

METHOD Pop returns integer
action
   integer return_val ;

   if top <= 0 then           /* Return -1 if stack empty. */
      return -1
   else                       /* Otherwise return top element. */
      begin
         return_val := Stack[top] ;
         Stack[top] := 0 ;
         top := top-1 ;
         return return_val ;
      end ;
end action ;

METHOD Clear_All              /* Clear all stack entries. */
action
   while top >= 0 do
   begin
      Stack[top] := 0 ;
      top := top-1 ;
    end ;
    top := 0 ;
  end action ;

  END.                    /* End of object declaration. */
```

Figure 4.2. General language features : Object Declaration.

While the basic structure of the model is adequately summarised by the above list of basic points and the example code of Figures 4.1 and 4.2, these descriptions actually represent nothing more than a skeleton to which more detailed concepts can be added. The remainder of this chapter will flesh out this basic framework, describing the use of active objects, language support for timing constraints and appropriate mechanisms for atomic actions and exception handling. However, an important point to note before proceeding is that the major purpose of the programming notation shown here is not to propose a "new" real-time programming language. Its main aim is to provide a vehicle for expressing the ideas upon which the real-time object model is based and it is the development and application of the model that is the major focus of this thesis.

4.2. Active Objects, Method Selection and Event Handling.

As well as responding to service requests from system operators or users, a real-time control system must also carry out processing independently of those requests, monitoring its environment and responding appropriately to external conditions. Typically, such independent monitoring activity takes the form of one or more cyclic tasks which read values from external sensors, process those values according to the present state and operating conditions of the system and produce appropriate driving signals for external actuators. For example, consider a quality control sub-system which monitors containers of liquid passing along a conveyor belt in a factory. If a defective container is detected, it must be removed from the belt. This sub-system is easily implemented as a cyclic task which reads a value from some sensor capable of detecting faulty containers. If the sensor value indicates that the current container is defective, appropriate actions can be initiated to have that container removed. This type of *closed-loop* activity is encountered across a wide range of real-time control applications and any programming system or computational model intended for real-time use should provide suitable mechanisms to support such functionality.

4.2.1. Processes vs. Active Objects.

In a process-based programming system, it is very easy to support continuously executing, cyclic tasks, since each task can be implemented as a separate, cyclic process. However, when working within an object-based or object-oriented framework, it becomes more difficult to provide an appropriate mechanism for repetitive closed-loop activities. In many object-based systems, objects are essentially passive entities, simply responding to invocation messages as and when they arrive. This execution model is ill-suited to cyclic tasks, since it does not allow objects to carry out independent processing and an alternative mechanism must be provided to drive cyclic activities. For example, one possibility is to allow application programs to be designed as a combination of objects providing services and processes providing cyclic control, but this should be regarded as a poor compromise. Being neither solely process-based nor solely object-based, it runs the risk of being poorly structured, inefficient and awkward to implement cleanly while not offering any great advantages over either individual computational model. For object-based real-time applications, a better solution is to support *active* objects. These are objects which contain an internal process or thread of control that may carry out processing independently of any invocations received by the object. In essence, active objects can be regarded as an object-based encapsulation of a process. However, they can be used to extend the normal process concept in a number of ways. First of all, an active object may still contain ordinary methods and interact with other objects in the usual fashion. By taking advantage of this, the internal process of an active object can be isolated from direct contact with other objects in the application, communicating instead by means of updates to local state information which is then accessed externally via one of the active object's methods. This can help to prevent failed or faulty processes propagating errors throughout the rest of the system. Secondly, there is no reason why an active object need encapsulate only a single thread or process. Multi-threaded objects can be constructed, each encapsulating a number of related threads which need access to the same local state information. This may be more efficient than having a collection of related, but logically independent, processes and it may also allow the application programmer to develop a more appropriate logical structure for the application software as a whole.

```
OBJECT Sensor IS

STATE
   real old, last, latest ;

METHOD Read_New_Values      /* Private method for reading  */
action                      /*     new sensor value etc.   */
   old := last ;
   last := latest ;
   latest := /* Get value from sensor hardware. */ ;
end action ;

METHOD Average_Reading returns real
action
   real average_value ;

   average_value := (old+last+latest)/3 ;
   return average_value ;
end action ;

METHOD Latest_Reading returns real
action
   return latest ;
end action ;

THREAD          /* Declaration of internal thread of control.   */
cycle                   /* (Several threads can be declared     */
   Read_New_Values ;    /*        if they are needed.)          */
end cycle ;

END.
```

Figure 4.3. Active object declaration.

Figure 4.3 shows a typical active object declaration. This is identical to the declaration for a passive object, but for the inclusion of declarations for internal processes or threads of control. An active object may contain more than one thread and a thread may perform one of three distinct functions :

1. *Object initialisation*

   All threads begin execution immediately when an active object is instantiated.

   A non-periodic thread can therefore be used to perform initialisation within the object. This thread will carry out its activities immediately upon creation of the object and then terminate.

2. *Background (Control) processing*

   Cyclic threads within active objects can be used to perform closed-loop control processing such as that described earlier. In most applications, this will probably be the most common use of active objects.

3. *Method Selection*

   In some applications, for example event-driven systems, it may be necessary for objects to restrict the order in which they accept invocations. An active object may contain one thread which performs this particular task, using a select/accept mechanism (see Figure 4.4). A more detailed description of this mechanism is given below.

```
select
   accept Request_1 → accept Request_2 ;
□
   accept Request_3 → /* Further processing. */
□
   delay(100ms) → /* Timeout processing. */
end select ;
```

Figure 4.4. Syntax of select and accept statements.

An important point to note is that in those objects which include a thread performing method selection (as described above), all incoming method invocations to the object are mediated and controlled by the method selection thread. However, such objects may still include other background threads that execute independently of the method selection thread. Also, in objects that do not employ method selection, incoming invocations are passed directly to the appropriate methods within the object, as in any other object-oriented programming system.

4.2.2. Active Objects and Method Selection.

The inclusion of the method selection mechanism shown in Figure 4.4 allows the implementation of sophisticated control algorithms in which future activity is dependent upon past events in the environment or past actions within the control system. For instance, consider the following simple sub-system, which might be implemented as a single object. There are four operations, A, B, C and D. Requests to execute these operations may arrive in any order, however there are only two valid execution sequences: A-B-C or D-C. Also, if the latter sequence of operations is executed, some kind of internal processing must follow the completion of C. If a request arrives that is not currently valid (e.g. request D immediately following A), then it should be queued until it can be served. In most object-based systems, ensuring that only the correct execution sequences were followed would be quite difficult, however if the object contains a thread limiting the set of invocations that it is prepared to serve, the problem can be solved quite neatly. Figure 4.5 shows a possible implementation of such an object. The select/accept mechanism is similar to that used in the Ada language. The use of **select** is limited to the non-deterministic selection of one of a number of **accept** statements, and **accept** can only be performed on an object's public (exported) methods.

```
OBJECT Example IS

STATE
   /* State variables. */

METHOD A (...)
action
   ...
end action ;

METHOD B (...)
action
   ...
end action ;

/* Declarations for methods C and D. */

THREAD
begin
   cycle
      select
         accept A → accept B → accept C ;
```

```
        □
          accept D → accept C → begin
                                    /* Further processing.  */
                                  end ;
        end select
      end cycle
    end

    END.
```

Figure 4.5. Use of method selection.

4.2.3. Method Selection and Event Handling.

Another area where the method selection mechanism proves to be useful is in structured event-driven applications. For example, consider the case where operations A, B, C and D of Figure 4.5 are handlers for four different external events that can occur concurrently. Further, impose the restriction that events should only be accepted and processed in the order A, B, C or D, C. The mechanism described earlier would allow such event sequences to be processed correctly, whether two or more of the events occurred concurrently or not.

```
Object_A.Method_1(...) ;      /* "Normal" synchronous      */
      ...                     /*   invocation.             */
      ...
^Object_A.Method_2(...) ;     /* Asynchronous  invocation. */
                              /* (Signified by leading ^). */
```

Figure 4.6. Asynchronous invocation.

Also, by using the method selection technique in conjunction with asynchronous communications (the syntax of which is shown in Figure 4.6), objects can be allowed to perform cyclic monitoring activities, where outgoing invocations are generated in response to local events while the object continues to carry out its monitoring function. This is illustrated more clearly in the call control example given in the next chapter.

4.2.4. Threads and Procedures.

A final point of note regarding active objects is that they can also contain definitions of "procedures". A procedure, in this context, is simply an encapsulated thread that can be called by any of the other threads within the object to perform some particular task.

Procedures are private, in that they cannot be called from outside the object. Neither can they be called by any of the object's methods, only by its internal threads. Hence, a procedure is, essentially, just a "sub-thread" that provides a specific operation and the inclusion of this particular construct is purely intended as a code-saving device for those situations where one or more threads need to carry out the same activity. An example of this can also be seen in the call control application given in the next chapter.

4.2.5. Concurrency Control in Active Objects.

Throughout all of the above, it has been assumed that concurrent threads within an active object obey the internal concurrency restrictions described in the previous chapter. That is, threads are either completely independent, or only interact through local or remote method invocations. Since it is assumed that all methods are executed as atomic actions, it follows that concurrency control for parallel threads will be provided by the concurrency control mechanisms that are used to enforce serialisability. Hence, it is ensured that interactions between concurrent threads within a single object will be suitably controlled.

4.3. Timing Constraints.

One of the most important aspects of any real-time programming system is the way in which timing constraints are handled. Since the notion of time is central to the real-time application domain, it is appropriate that time should be treated as a "first-class" entity in any real-time programming language or programming model. There are several ways in which this can be achieved. For example, in the CHAOS system, any invocation may have real-time constraints (start time, deadline etc.) associated with it. An alternative approach, taken in the MARUTI system and also adopted here, is to provide language constructs for timing constraints. In MARUTI, a range of primitives are supported including `every`, `after`, `before`, `at` or `within`. However, the model proposed here relies upon only two such mechanisms - one for periodic tasks and one for aperiodic (sporadic) tasks - illustrated in Figure 4.7. Used appropriately, these two constructs can subsume the functionality of the individual timing primitives used in the MARUTI system, as illustrated in the examples given in Figure 4.8.

```
every 50ms do          /* Periodic temporal scope.             */
  lead_time (5ms) ;    /* Initial delay before execution.      */
    Operation1 ;
      ...              /* Code for operations.                 */
    OperationN ;
  lag_time (7ms) ;     /* Required free time at end of period. */
end every ;

do                       /* Aperiodic temporal scope.   */
  Operation1 ;
    ...                  /* Required operations.        */
  OperationN ;
start_in (5s,7s) ;       /* Start interval. (Relative)  */
finish_in (15s,20s) ;    /* Finish interval. (Relative) */
```

Figure 4.7. Periodic and aperiodic temporal scopes.

The major reason for adopting a language-level approach to timing constraints, rather than an approach based solely upon placing timing constraints on method invocations is that it allows timing constraints to be specified for groups of operations without having to encapsulate those operations as a particular method within some object. Also, it helps to make timing constraints more explicit within the structure of an application and, as will be seen later, allows individual timing exceptions to be supported in a clear and direct manner.

```
    at t do S ;                   within t do S ;

  can be expressed as           can be expressed as

    do                            do
      S ;                           S ;
    start_in(t,t) ;               finish_in(t,t) ;
```

Figure 4.8. Expressing different timing constraints. (Examples.)

With regard to scheduling, in a dynamic system, the timing primitives that have been adopted would be used to pass task scheduling information to the underlying run-time scheduler provided by a real-time operating system, while in a static system, they would provide a means of detecting and trapping unexpected timing errors.

4.3.1. Time-base : Absolute and Relative Times.

For most real-time applications, it is reasonable to assume that all objects have access to a global time-base that will allow them to determine the current time (relative to some known reference) and measure the passage of time intervals with some known, bounded accuracy. In a centralised, single-processor system, such a time-base can be provided by the system clock, while processing nodes in a distributed system can have local real-time clocks that are synchronised to some known level of accuracy using an appropriate clock synchronisation algorithm (see, for example, [Halpern 84][Lamport 85][Srikanth 87]).

If an appropriate time-base is available, timing constraints can then be expressed using either absolute or relative times. For instance, if some event occurs at time τ according to an object's local clock value and it initiates an action which must be completed within 100 ms, then the deadline for that action could be expressed as 100 ms (relative time) or as τ+100 (absolute time). These different means of expressing the same constraint each have their own advantages and disadvantages. While absolute times are most useful for synchronising internal system actions with forthcoming external events (e.g. action A must complete by 6:00), relative times are best suited to periodic tasks (e.g. repeat task T every 10s) or responses to external events (e.g. given than event E has occurred, a response must be generated within 5s). Also, when mapping timing constraints between objects during the course of an invocation, the use of relative times requires that allowances be made for communications delays at both the caller and the callee, while the use of absolute times restricts such allowances to the callee but requires that an adjustment be made for clock synchronisation.

For generality, it is assumed that a synchronised clock service is available and the language constructs shown in Figure 4.7 can be used for either absolute or relative timing constraints by adopting the convention that constraints given with units (seconds, milliseconds, etc.) represent relative times, while those given without units refer to absolute times.

4.3.2. Timing Errors.

To deal with timing errors, built-in timing exceptions are associated with both types of temporal scope. Six exceptions are supported and these are listed, along with their meanings, in Figure 4.9. The raising of these exceptions and the provision of appropriate exception handlers is discussed in more detail in the section on exception handling later in this chapter.

Exception.	*Meaning.*
`Early_Start`	Aperiodic task started too early. Periodic task started too early. (Delay in execution at beginning of period too short.)
`Late_Start`	Aperiodic task started too late. (Missed its start time.)
`Early_Finish`	Aperiodic task finished too early.
`Late_Finish`	Aperiodic task finished too late (Missed its deadline.) Periodic task finished too late. (Specified free time not available at end of period.)
`Period.`	Periodic task missed period completely.

Figure 4.9. System timing exceptions.

4.3.3. Propagation of Timing Constraints.

Invocations made from within a timing constraint have an implicit parameter associated with them giving the caller's current deadline. If the called operation does not have its own timing constraint, this deadline information is used to calculate a latest finish time for the callee, allowing for communications delay in its reply to the caller. Typically, if the callee cannot meet this implicit timing constraint, it will perform local state restoration and the responsibility for external error recovery will lie with the caller. This seems to be a sensible strategy to adopt, since :

i. it would be difficult to ensure that the callee performed an appropriate local recovery operation unless one had been specified as part of a local timing constraint, and

ii. the violation of the timing constraint at the callee implies that the original constraint at the caller will also be violated and the caller will, therefore, already be committed to performing error recovery.

4.4. Atomic Actions.

While atomic actions offer an excellent structuring technique for fault tolerant distributed applications, the use of a traditional transaction mechanism in a real-time environment would suffer from a number of disadvantages. In particular, a transaction system that relied solely upon the use of backward (state based) error recovery would be unsuitable for use in a real-time system. Since real-time control software interacts with an external environment, it may be impossible to restore a system to an earlier state. Even where state restoration is feasible, its use may cause deadlines to be missed, since it does not serve to further the current activity of the control system.

4.4.1. Operation-Based Recovery : Application Specific Abort.

What is needed is a method by which atomic actions may be aborted using programmer defined, application specific recovery operations, similar to the anti-operations used in the CHAOS system. For example, consider the following application. A robot manipulator must pick up a gear-wheel from a moving conveyor belt, align it with a spindle on another conveyor belt and lower the gear into place. This operation must be performed every five seconds. If the operation is regarded as an atomic action and, for some reason, the manipulator cannot position the gear-wheel correctly, then there is no easy way to restore the state of the system and re-attempt the action. The gear-wheel cannot simply be returned to its own conveyor belt, since the belt has continued moving. However, assume that there is a return belt for unused gears and spindles. An obvious (and simple) recovery action for the manipulator would be to place the "faulty" gear-wheel and spindle onto the return belt, pick up the next gear-wheel and attempt to fit that to the next spindle. It is likely that this recovery action could be performed in a sufficiently short time not to jeopardise the timing of the system as a whole and the returned gears and spindles could either be re-directed onto the appropriate conveyors or checked for defects by some quality control system. By allowing this type of application specific recovery, the failure atomicity property of atomic actions can be preserved without sacrificing efficiency at the application level.

For maximum flexibility, objects are allowed to possess a number of abort types (as shown in Figure 4.10). All operations are executed as atomic actions and any operation may call any of the available abort types, according to requirements. Further, an operation may call different abort types depending upon its internal state and the type of error that occurs. For example, if an error is detected early during execution, then state-based recovery may be appropriate, whereas at a later point, operation-based (forward) recovery may be more suitable.

```
OBJECT Abort_Example IS

STATE
  /* State variables. */

ABORT
  ABORT_TYPE_1  :begin
                   /* Code for application specific */
                   /*   recovery operation.         */
                 end ;

  ABORT_TYPE_2  :begin
                   /* Other recovery operation.     */
                 end ;

  /* Definition of other abort types if required.*/

  /* Declaration of methods, threads etc. as normal.  */

END.
```

Figure 4.10. Declaration of abort types.

4.4.2. Application Specific Commit.

A logical further extension to this mechanism would be to allow atomic actions to carry out application specific commit operations (see Figure 4.11). There may be some activities in real-time control applications which do not require their state to be committed to stable storage when they complete. At one extreme, some functions may not require an explicit commit operation (i.e. a null commit), while, at the other, the required commit operation may involve a complex interaction with the system's environment. In such circumstances, the ordinary state-based commit processing used in many transaction systems would be inappropriate.

```
OBJECT Commit_Example IS

STATE
   /* State variables. */

COMMIT
   COMMIT_TYPE_1 :  begin
                       /* Code for application specific */
                       /*   commit operation.           */
                    end ;

   /* Definition of other application specific commit  */
   /*   operations (if required).                       */

ABORT
   /* Definition of Abort types. */

/* Normal Method and Thread declarations etc.   */

END.
```

Figure 4.11. Declaration of commit types.

4.4.3. Definition and Use.

The definition and use of these application specific commit and abort types is illustrated in Figure 4.12. For actions that require them, the normal state-based commit and abort mechanisms are provided as a default and called by omitting the commit or abort type specifier in the commit or abort command.

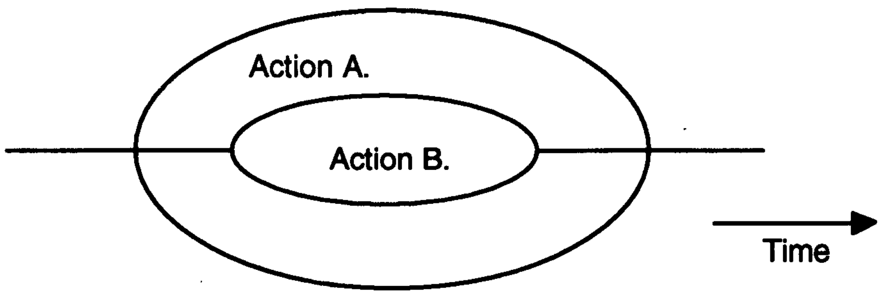
```
OBJECT Launch_Controller IS

   /* Simple defensive missile launcher. (Very unrealistic, */
   /*   but it illustrates the major points very well.)     */

STATE
   /* State variables. */

COMMIT
   ARM  :  /* Arm missile. */
   FIRE : /* Send launch signal to hardware. */

ABORT
   DISARM :     /* Disarm missile. */
   DETONATE :   /* Send abort (detonate) signal to missile. */
```

```
METHOD Acquire_Target (IN Coords T)
action
   /* Engage tracking radar to acquire target. */
   if NO_TARGET then
      Abort ;              /* State-based (default) abort. */
end action : Commit(ARM) ;

METHOD Confirm_IFF
action
   /* Check target identification, Friend/Foe. */
   if TARGET_FRIENDLY then
      Abort(DISARM) ;
end action : Commit(FIRE) ;

METHOD Track_Launch
action
   /* Engage tracking radar for outgoing missile. */
   if MISSILE_FAULT then
      Abort(DETONATE) ;
end action : Commit ;       /* State-based (default) commit. */

THREAD
   begin
      cycle
         accept Acquire_Target →  begin
                                     Confirm_IFF ;
                                     Track_Launch ;
                                  end ;
      end cycle ;
   end ;

END.
```

Figure 4.12. Using programmed commit and abort operations.

4.4.4. Nesting.

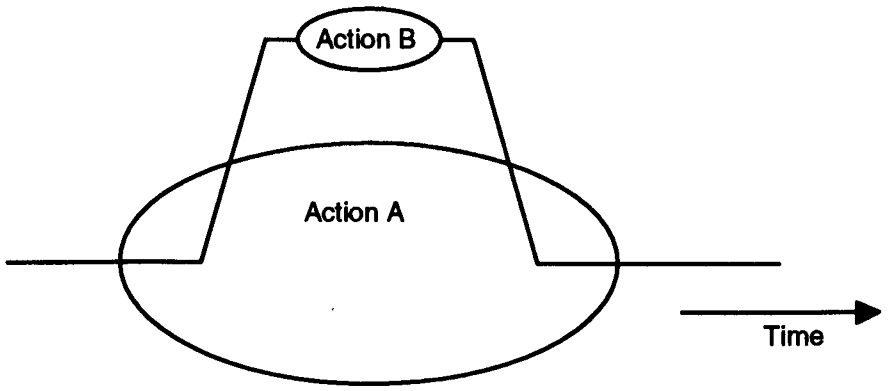
As described in chapter 2, atomic actions can be nested (see Figure 4.13), in which case the effects of the nested action (B) do not become permanent until such time as its parent action (A) commits. Further, if the nested action commits, but its parent action subsequently



Action B called from within action A.

Figure 4.13. Nested atomic actions.

aborts, then the effects of the nested action must also be undone. Such nesting can be common in transaction based systems, however it can cause problems in a real-time environment. The issue of performance must be considered, since a nested action cannot complete its execution (from the point of view of the rest of the system) until its parent action completes and the parent action, while running, must retain exclusive access to all resources that were used by the nested action, even if they are no longer required. The other major problem with nested actions in real-time systems is that situations can arise where the effects of a nested action are immediately visible throughout the system because the action has performed some operation that has an effect upon the system's external environment. For example, consider a nested action in a flight control system that causes an engine to be shut down. Such an action may well be nested with respect to the logical structuring of actions in the application, but its effects will be instantly visible to the rest of the system and there is no way for its parent action to prevent this.



Action B executed as top-level action from within action A.

Figure 4.14. Nested top-level atomic action.

In real-time applications, actions that can have an effect on the external environment of the system are best treated as being nested top-level actions (Figure 4.14), since their external effects may be observed by other ongoing actions. This approach would seem to be more sensible than attempting to ensure exclusive access to sets of external devices or attempting to derive nesting information for actions that interact through agencies external to the computer system. The potential loss of concurrency that might result from either of these alternative strategies would typically be much more problematic than the restriction to top-level actions. However, correctly identifying those actions that can cause visible external effects and enforcing a strict division into actions that can be properly nested and those that cannot may also prove to be difficult. An easier and more tractable approach is to impose the restriction that all operations are performed as top-level actions, regardless of their logical nesting. This does not have an impact upon the expressive power of the object model (as the next chapter will demonstrate), but it does help to ensure that actions with external effects are handled properly and it allows the outputs from an action to be made available to the rest of the system as soon as the action commits.

4.4.5. Serialisability and Distributed Actions.

Another aspect of the atomic action mechanism that must be examined more closely if actions are to be used in a real-time environment is the requirement for serialisability. A number of methods have been proposed to ensure that concurrent atomic actions do not share data and become interdependent, since this can lead to situations where cascading aborts may take place (see Section 2.3.1). However, many of these serialisability mechanisms do not take any account of the timing constraints or relative priorities of the actions that they affect. For example, in ordinary two-phase locking schemes, once an action has acquired a lock, other actions are forced to wait for that lock to be released, even if they are of higher priority or more urgent than the action currently holding the lock. Similarly, when conflicts occur under optimistic concurrency control schemes, high priority actions may be forced to abort when conflicts occur with lower priority actions. In real-time applications, alternative concurrency control strategies must be adopted that take account of the priority and urgency of actions. A number of research efforts are already under way in this field and real-time concurrency control mechanisms such as those described in [Wolfe 91] or [Haritsa 90] could be used to support serialisability in real-time transaction-based systems.

Finally, an important point to note with regard to the atomic action scheme proposed here is that actions are not permitted to be distributed. In some systems, a single atomic action may encompass activity in several objects and, in distributed systems, this may involve execution at several different processing nodes. This situation can arise when an action invokes nested actions at other objects or when an action invokes other remote operations that are not themselves atomic actions. In the latter case, the remote operation will fall within the scope of the invoking action for the purposes of concurrency control and recovery. This can complicate the serialisability mechanisms that are required and it may lead to a degradation in performance of the system as a whole. However, imposing the restriction that all operations are performed as top-level actions again serves a useful purpose in dealing with this particular problem. By effectively prohibiting distributed actions, it confines all concurrency control decisions to the scope of individual objects and makes it feasible to employ a simpler concurrency control mechanism (for example, simple mutual exclusion within objects). The restriction to local, top-level actions also makes it unnecessary to execute a distributed two-phase commit protocol when actions terminate. This is another important feature for real-time applications, since a distributed two-phase commit can take a considerable time to execute and, in the worst case, can be non-terminating in the presence of failures.

4.5. Exception Handling.

While the use of an atomic action mechanism can help to maintain consistency within individual objects and prevent the flow of erroneous information between objects, situations may arise where errors or failures cannot be handled properly within the scope of a single object. Under such circumstances, an object may need to return some form of error indication to its caller so that error recovery can be performed at a higher level within the system. Normally, in a transaction based system, the only such indication that the caller would receive would be a signal that the called action had aborted. This is very much in keeping with the program structuring concepts that the atomic action embodies, but it runs the risk of allowing inefficient or, in the worst case, inappropriate error recovery in the caller. If a more specific error indication could be returned from the failed operation, more efficient and appropriate error recovery could be provided. One of the best ways to support such functionality would be by means of an exception handling mechanism. By allowing exceptions to be raised when an action aborts, the caller of that action can be given some indication of the nature of the error that has occurred. Further, in this case, the exception handling mechanism can be made relatively simple. A single-level, termination mechanism is most appropriate and, since error recovery within the called object would be performed as part of the atomic action abort operation, there is no need to make large amounts of local state information available to the caller. So, in essence, exception handling is mainly used as a structured return code mechanism, hence its simplicity.

```
METHOD Safe_Div (IN real dividend, divisor)
  returns real
  SIGNALS Div_by_Zero         /* Declare exceptions that   */
action                        /*   may be raised.          */
  if (divisor=0) then
    signal Div_by_Zero          /* Raise exception.   */
  else
    return dividend/divisor ;
end action ;

  C = Safe_Div(a,b) [ Div_by_Zero :  begin
                                       /* Code to handle   */
                                       /*   exception.     */
                                     end ] ;
```

Figure 4.15. Exception and exception handler syntax.

The syntax which has been adopted for exceptions and exception handlers is based on that used in [Cristian 82] and is shown in Figure 4.15. Whenever an action aborts, an exception is raised to the caller. This may be a user-defined exception, raised using a `signal` statement, or the default exception (`Fail`) which is raised automatically whenever actions

```
METHOD Default_Handler_Example
action
    Operation1(x,y,z) [ Exception1 : begin...end ] ;
    ...
        /* Code for remainder of method, including other  */
        /*   specific exception handlers where required.  */
    ...
end action : Commit : [ begin
                          /* Code for default      */
                          /*   exception handler. */
                        end ] ;
```

Figure 4.16. Default exception handler.

abort without raising any user-defined exception. When calling an operation, the caller must provide a handler for any exception that might be raised, with the proviso that two or more exceptions may share the same handler and the caller may also provide a default handler to catch any exceptions not explicitly covered (see Figure 4.16).

There is no mechanism to support the implicit or direct re-raising of an exception, however, where such functionality is required, an exception handler may itself explicitly raise an exception using a signal statement and the same exception name may be re-used at the application programmer's discretion (see Figure 4.17). Finally, exception handlers may be nested. Although it can be argued that the excessive use of nested exception handling points to flaws in the design or structure of an application program, there may be occasions when it is necessary within an exception handler to call operations that might themselves raise exceptions. It would, therefore, seem to be best to allow the nesting of exception handlers if the application requires it.

```
METHOD Re_Raise_Example
  SIGNALS B_Exception
action
  Operation_A(...) ;
  Operation_B(...) [ B_Exception : signal B_Exception ] ;
end action : Commit
```

Figure 4.17. Re-raising an exception by re-use of exception name.

4.5.1. Timing Exceptions.

As mentioned earlier, the temporal scope constructs used to express timing constraints provide a set of default timing exceptions that can be used to trap timing errors. When a timing error occurs, the appropriate exception is raised *by the temporal scope statement that expressed the violated timing constraint.* So, for example, exceptions such as `Early_Start` or `Late_Start` are raised by the `start_in` or `lead_time` statements, while exceptions relating to a task's finish time are raised by the `finish_in` or `lag_time` statements. Similarly, the `Period` exception can only be raised at the end of a periodic temporal scope. This assignment of exceptions to specific parts of the temporal scope constructs helps to maintain the logical correspondence between a given exception and the timing constraint to which it relates. It also helps to break down what might otherwise be a lengthy and complex exception handling block at the end of each temporal scope.

```
every 50ms do
  lead_time (5ms)  [ Early_Start : ... ] ;
    Operation1 ;
      ...
    OperationN ;
  lag_time (7ms)   [ Late_Finish : ... ] ;
end every      [ Period : ... ] ;

do
  Operation1 ;
    ...
  OperationN ;
start_in (1s,3s) ; [ Early_Start : ... ; Late_Start : ... ] ;
finish_in (5s,9s) ; [Early_Finish : ... ; Late_Finish : ... ] ;
```

Figure 4.18. Syntax and use of timing exceptions.

Precise meanings for each of these exceptions were given in the table of Figure 4.9. It should also be noted that timing exceptions are special in that the applications programmer is not always obliged to provide handlers for all timing exceptions. In particular, the `Early_Start` and `Early_Finish` exceptions may be disregarded, in which case tasks that start early would simply be suspended until their earliest start time, while tasks that end early would be prevented from returning their results until after their earliest finish time.

The major advantage of using exception handling techniques to deal with timing errors is that it allows application specific recovery actions to be performed when tasks violate their timing constraints. The application programmer is therefore free to implement the most appropriate and efficient recovery operations for any given task failure. In a real-time environment, this is a highly desirable feature, since the violation of a timing constraint may lead to a subsequent catastrophic failure of the whole system and appropriate recovery measures should be effected as rapidly as possible.

4.6. A Summary of the Object Model.

It will be worthwhile at this point to summarise the features that are offered by the object model that has been proposed and to consider the advantages that it might offer the real-time applications programmer. As a starting point for this summary, consider the following features that were originally listed in section 3.1 :

1. *Objects are of medium granularity.*

   Typically, application objects will correspond to specific devices, device controllers or control sub-systems. This not only offers a good trade-off between the advantages and disadvantages of the object-oriented approach, it should also allow the application programmer to mirror the physical structure of the real-world system in the logical structure of the control software.

2. *Objects may be active*

   Objects may possess independent, internal threads of control. This provides support for continuous cyclic activities without having to resort to a separate process-based primitive.

3. *Objects may communicate via synchronous or asynchronous invocations.*

   Although the synchronous model of communication is common in many object-based systems, a corresponding asynchronous mechanism should be provided since it may sometimes be more appropriate for the real-time environment. This is particularly true in event driven systems or for interactions where one object only needs to notify another of some condition (e.g. alarm conditions).

4. *An object is only permitted to have a single interface.*

   This restriction avoids problems with the management of multiple interfaces and allows communications between objects to be more rigidly defined.

5. *All operations are performed as atomic actions.*

   By executing all methods as atomic actions, fault tolerance can be provided both within objects and across invocations between objects. The non-interference (serialisability) property of the atomic action is also useful if objects are to be considered as extended state machines for replication (see below).

These five basic properties serve to define the general structure of objects and their methods, however four other specific features that help to provide support for real-time applications can now be added to the list:

6. *All atomic actions are executed as top-level actions.*

   Executing all operations as top-level atomic actions, even if they are logically nested, has three major advantages in a real-time environment. First, operations that have external effects on the system's environment cannot be treated properly as nested actions, so the restriction to top-level actions ensures that such operations can always be handled correctly. Secondly, the blanket use of top-level actions simplifies some of the concurrency control problems associated with serialisability by prohibiting complex distributed actions. Finally, the outputs from a top-level action can be made available to the rest of the system as soon as the action commits and the resources or locks that it was holding can be released for use by other actions.

7. *Application specific recovery mechanisms.*

   By allowing atomic actions to perform programmed commit and abort operations, application specific error recovery can be supported. More efficient and flexible recovery operations can therefore be provided, taking into account real-time constraints, interactions with external devices and system safety constraints. The provision of an exception handling mechanism further facilitates error recovery, since it allows a failed operation to pass specific error indications back to its caller.

8. *Support for timing constraints.*

   The notion of time and timing constraints is central to the real-time applications domain. The provision of both periodic and aperiodic temporal scope constructs at the language level allows real-time constraints to be directly expressed within the application software. Built-in timing exceptions can then be provided to allow timing errors to be trapped and handled immediately at their point of occurrence.

9. *Mode changes and event handling.*

   A method selection mechanism is provided for active objects. This serves two separate purposes, allowing complex execution paths to be specified for an object (e.g. mode changes) and providing support for event-driven applications (since events can then be mapped to method invocations and the set of acceptable events at any given time limited by method selection.)

Objects constructed according to this model are well-suited to the real-time applications domain. The temporal scope constructs allow timing constraints to be specified very easily and the atomic action and exception handling mechanisms allow the programmer to provide application specific recovery operations for timing and other errors. System mode changes and event handling can both be supported, while active objects provide an effective mechanism for closed loop, cyclic processing. However, none of the features described so far have any bearing on the run-time *availability* of objects. Availability requirements can only be met by employing some form of object replication scheme and this, in turn, places constraints upon any computational model that is to be used. Care has therefore been taken to ensure that objects constructed according to the model shown here can be replicated using any type of replication strategy.

4.7. Supporting Object Replication.

The choice of object replication strategy for a given application will depend upon a number of factors: the types of fault that are to be tolerated, available resources within the system, performance constraints, etc. It will also depend upon the computational model that underlies the objects themselves. As explained in chapter three, any replication strategy embodies a set of assumptions about the communications support provided by the system and the failure modes and computational properties of individual replicas. Hence, if a particular replication strategy is to be used for some application, appropriate constraints must be placed upon the computational model used in the application software. Alternatively, if the computational model of an application is already fixed, it may be impossible to employ certain replication techniques in that application.

Passive replication schemes, although they rely upon replicas being fail-silent, do not impose any restrictions upon basic computational properties and there are no major restrictions on the types of object that can be replicated using passive techniques. The impact of passive replication on the object model proposed here is, therefore, very small. Active replication schemes, on the other hand, impose the constraint that all computations must be deterministic and it is this particular restriction that has guided the development of this object model.

4.7.1. Objects as State Machines.

In some ways, the object model proposed here is similar to the state machine model described in the previous chapter. Like state machines, objects consist of internal state variables and a set of commands (the object's methods) which transform that state and produce outputs. Further, the abstract process of invoking a method closely mirrors the execution of a state machine command and, since all operations are executed as atomic actions, the execution of any given method is atomic with respect to the execution of all others. However, where one of the defining characteristics of a state machine is that it specifies a deterministic computation and is, therefore, amenable to active replication, the objects described here can include several potential sources of non-determinism. This would seem to make such objects unsuitable candidates for any active replication technique, but this is not, in fact, the case.

Although objects may contain internal threads of control, set timing constraints and interact with their environment, the techniques described at the end of the previous chapter can be used to ensure that the execution of such operations is deterministic from the point of view of an object replica group. It has been shown that the `select...accept` mechanism used for mode changes and event handling can be implemented in such a way as to allow non-deterministic selection to be agreed between replicas. This mechanism can then be used at a lower level to provide a non-divergent implementation of the temporal scope constructs. Internal threads, either in active objects or created within a method by use of the `par...end par` mechanism, can be handled deterministically by imposing the restriction that they only interact or manipulate state information by means of message passing or method invocations. Finally, interactions with an external environment can be mapped to appropriate method invocations. Given these mechanisms, objects can be regarded as an extended form of state machine for the purpose of replication and the use of an appropriate communications protocol (e.g. atomic broadcast) for interactions between objects and object groups will ensure that the states of different replicas do not diverge.

4.8. Comparison with Existing Models.

To conclude this chapter, it will be interesting to compare the object model described here with two other existing real-time object models: the model proposed by Kopetz and Kim in their analysis of temporal uncertainty [Kopetz 90] and the model adopted in the ARTS real-time system [Mercer 90]. In each case, a brief description of the appropriate object model will be given, followed by a short discussion of the way in which the model shown here relates to that existing model.

4.8.1. The Object Model of Kopetz and Kim.

The model of Kopetz and Kim is largely concerned with the behaviour of distributed real-time control systems in the time domain and preventing or minimising the uncertain timing behaviour of such systems. First of all, it is assumed that the local clocks of the nodes within a system are synchronised to some known precision π and the granularity (i.e. time between successive ticks, as measured by some external reference clock) of this synchronised timebase is assumed to be g, where a value of g is chosen such that:

$$\pi < g < 2\pi.$$

This is to ensure that globally meaningful time stamps can be generated. The concept of a *real-time entity* (RT-entity) can now be introduced. This is something of relevance to the system's purpose, which contains a time-varying internal state. For example the temperature of a particular vessel, the speed of a vehicle, etc. Since the state of such an RT-entity changes with the progression of time, it can only be represented within an object-based framework if the classical notion of an object is extended to include knowledge about real-time. A *real-time object* (RT-object) is therefore defined to be an object, O_k, which has a synchronised clock C_k of granularity $g_k >> g$ associated with it.

The clock C_k defines a synchronous time-grid for the object O_k and the granularity of this time-grid is chosen to agree with the dynamics of the RT-entity that O_k represents. At every tick of the clock C_k, a message is sent to one of the methods of the RT-object. This may be the only message that can activate the RT-object, in which case the object is *synchronous* and all other incoming messages are queued until they are polled by the procedure that is activated by the clock message. Alternatively, an RT-object may be *asynchronous*, in which case any message can activate one of its methods and the clock message will result in a null action most of the time. In either case, there is a *state visibility constraint* that restricts the external visibility of the internal states of the RT-object to those states that are occupied at points on the object's time grid.

4.8.1.1. Comparison.

The real-time object model that has just been described was developed as part of a larger, system-level model that attempts to support reasoning about the consistency and accuracy of real-time data and about the performance of real-time communication protocols. The object model proposed in this thesis, on the other hand, is entirely geared towards the programming of real-time applications software. The main result of this fundamental difference in purpose is that Kopetz and Kim's model is *time-triggered*, with actions being initiated in response to the passage of time, whereas the model proposed here is *event-triggered*, with actions being initiated in response to events in a system's environment. The time-triggered approach is particularly well-suited for analysis, since it essentially represents a static system in which events are handled at pre-scheduled times. Conversely, the event-triggered approach corresponds to dynamic systems, thus gaining a greater degree of flexibility at the expense of being less amenable to analysis.

However, in spite of differences in their basic purpose and underlying structure, there are some interesting parallels that can be drawn between the two models. For example, RT-objects essentially provide the same abstraction as active objects. In a sense, synchronous RT-objects correspond to those active objects that employ method selection to control their execution or that contain an internal thread but no public methods. Similarly, asynchronous RT-objects correspond to the more general class of active objects that contain both internal, cyclic threads and freely-accessible public methods. Also, the object model proposed in this thesis includes a mechanism that is equivalent to the state visibility constraint of Kopetz and Kim's model. Specifically, since all methods are executed as atomic actions, the state of an object will only be visible to other objects at those times when it is known to be consistent.

4.8.2. The ARTS Real-Time Object Model.

ARTS is a distributed real-time operating system designed for a real-time systems testbed being developed at Carnegie-Mellon University [Tokuda 89]. As part of the ARTS project, a real-time object model has been developed that includes a time-fence protocol which is used at every invocation in an object to detect the origin of timing errors. Every computational entity in the ARTS system is represented as an object, called an "artobject", and each operation that an artobject provides can be associated with a worst case execution time ("time-fence") value and a time exception handling routine. If an operation is called and its worst case execution time is greater than the remaining execution time for that operation at the caller, the operation is aborted as a time fence error.

```
class Sample_Artobject          // Specification.
{
  type private_data_object ;
  type abort_opr1() ;           // Note : These are timing
  type abort_opr2() ;           //   exception handlers
  Thread Thread_Root() ;
  ...
public:
  type opr1(...) ;   //# within time except abort_opr1()
```

```
        type opr2(...) ;    //# within time except abort_opr2()
        ...
      }

      // Class Sample body.
      //
      Thread Sample_Artobject::Thread_Root()   // Active object.
      {
        Accept(&invocation_dsc, &req_msg_dsc) ;
        ...
        DoComputation(&req_msg_dsc, &rep_msg_dsc) ;
        ...
        Reply(&invocation_dsc, &rep_msg_dsc) ;
      }
      type Sample_Artobject::opr1(...)
          //# within time except abort_opr1()
      {...}

      type Sample_Artobject::opr2(...)
            //# within time except abort_opr2()
      {...}
      type Sample_Artobject::abort_opr1(){...}
      type Sample_Artobject::abort_opr2(){...}
      ...
```

Figure 4.19. Example artobject declaration in ARTS/C++.

An artobject can be passive or active, in which case it may contain one or more internal threads of control and the designer of the object is responsible for providing appropriate concurrency control between concurrent operations. Active objects can also contain a *root* thread that is created and run immediately when a new instance of the object is created. Threads within artobjects can be defined to support periodic or aperiodic tasks and their timing attributes can include a value function as well as worst case execution time, period, phase and delay parameters. The objective of all these different features is to support *time encapsulation* among real-time objects and to bound timing errors at every object invocation. An example of an artobject declaration, written in the ARTS/C++ language (see also [Ishikawa 90]), is shown in Figure 4.19.

4.8.2.1. Comparison.

Like the real-time object model described in this thesis, the ARTS object model is intended to be a programming model rather than a model for analysis. The two object models therefore share many common features. For instance, both include mechanisms for expressing timing constraints and dealing with timing errors and both models allow objects to be passive or active and allow active objects to have multiple threads. Also, both models are event-triggered rather than time-triggered. However, there are also some important differences. The ARTS object model, like the CHAOS system, places timing constraints upon an object's operations, whereas the model proposed in this thesis takes a similar approach to the MARUTI system and uses a language-level timing primitive. In terms of fault tolerance, the ARTS model only provides recovery for timing errors, while the model described here provides more general fault tolerance coverage by means of the atomic action and exception handling mechanisms. Also, the ARTS object model has not been designed to allow for active replication and issues relating to event handling and operational modes have largely been ignored. These differences reflect a difference in the emphasis of the two models. ARTS is concerned with scheduling issues, at all levels from application programming down to basic operating system mechanisms, and the ARTS object model only represents a small part of the ARTS project as a whole. The model proposed here, on the other hand, has concentrated solely upon application level requirements and it is specifically geared towards supporting fault tolerance in real-time application programs. The end result is that the two object models differ in their scale and in their view of the programming problem, but not in their basic philosophy.

4.9. Chapter Summary.

This chapter has presented a model for fault-tolerant real-time objects. The general features of the object model were described, including the need to provide support for active objects, before moving on to consider mechanisms for expressing timing constraints and the provision of fault tolerance in real-time programs. An extended form of atomic action was illustrated, allowing application specific commit and abort operations to be defined, and the use of exception handling with this transaction-based mechanism was described. The use of system-defined timing exceptions to provide application specific recovery for timing errors was also described.

After summarising the overall structure of the object model, the use of object replication was discussed and it was shown that objects constructed according to this model can be regarded as extended state machines for the purposes of replication. The chapter then concluded with a brief examination of two other real-time object models, including brief comparisons of these existing models with the new model that had been described earlier.

Chapter 5.

Application Examples.

In the last chapter, a model for real-time objects was presented. A programming notation based upon this object model was also illustrated. The use of programmed commit and abort operations was shown, along with mechanisms for expressing timing constraints, raising and handling exceptions and managing active objects. Throughout all of this, care was taken to adopt a computational model and associated set of language constructs and support mechanisms that would allow such real-time objects to be replicated using state machine based active replication techniques. This, naturally, has imposed certain constraints on the object model and restricted the scope of activity for real-time objects. However, in this chapter, it will be shown that the general applicability of the object model to a range of different real-time applications has not been compromised as a result of these restrictions. This will be demonstrated by giving outline implementations of a number of different applications.

The examples that are to be used consist of a call-control system, two different robot control applications and a larger, train control application intended for use on a digitally-controlled model train layout. Each example will be introduced by a brief description of the system under consideration, before moving on to give an outline of a possible implementation in the programming notation described in chapter 4. At the end of each example, the general features of that application will be discussed, with regard to those features within the object model that are most useful in supporting applications of that type.

At this point, it must also be emphasised that the examples shown in this chapter are purely intended to illustrate the flexibility of the techniques and mechanisms adopted in the proposed object model. In particular, it must be understood that the development of the model thus far has been geared towards the practical aspects of real-time applications programming. Theoretical issues relating to analysis and verification, either of the model itself or of individual applications, are not considered in this thesis and it is not possible, at the present time, to prove specific properties (e.g. safety or liveness) for any of the example applications. The examples should, therefore, be regarded as proof of principle in the practical, rather than the theoretical or formal, sense.

Since the detailed code for each application would, typically, be quite long and complex, the implementations shown are outlines, in the sense that they omit some of the unnecessary detail associated with the complexities of internal algorithms and device-handling mechanisms. Before moving on to show the first example, the precise nature of these omissions will be discussed to show that they do not have an impact upon the applicability of the object model, or the expressivity of the programming notation that is to be used.

5.1. Subsidiary Algorithms and Device Handling.

The omission of unnecessary detail in the examples that are to follow has largely been restricted to two specific cases. The details of some of the internal algorithms upon which the main control algorithms of the system depend and details regarding the interface between the application software and the underlying hardware, specifically devices such as sensors, actuators, etc. In neither of these cases would the omitted code contain language constructs or mechanisms that violate any of the assumptions of the object model. Neither would it contain code providing access to functions that application objects would otherwise be unable to perform, although in the case of device drivers or device handling routines it would necessarily include calls to system and hardware specific library code. To illustrate this, two short examples (taken from the application examples) are given below.

5.1.1. Subsidiary Algorithms.

In many cases, the main control algorithms of an application rely upon subsidiary, support algorithms to provide some required function, typically involving the manipulation of local state information or some numerical calculation. By and large, the required algorithms are straight-forward computational processes or standard control operations and they have been omitted from the examples purely because they may, in some cases, prove to be lengthy sections of code. For instance, consider the calculation of alternative routes for blocked trains in the train control example. Basically, this would consist of searching through the track layout information for a branching point at which the blocked train could be diverted. A route from that branch point to some other, from which the train could resume its interrupted journey, would then have to be found. The conceptual simplicity of this problem hides the fact that the program code required for this operation would probably be quite long and depend upon the level of detail in the track layout information maintained by the track controller. If a list of pre-calculated diversionary routes was available to the controller, then finding an appropriate alternative route would only entail a table lookup operation or two. On the other hand, if no such list was available, then each diversion would have to be calculated on demand and an appropriate algorithm would have to be used that allowed diversions to be found as quickly as possible. A third possibility would be some form of monitor system that calculated diversions whenever accidents or blockages occurred and stored that information until the problem was cleared. Then, any train affected by the blocked sections of track could be informed of the appropriate diversion as quickly as possible. It has been assumed that this is the type of solution that has been adopted in the train set control example, with the track controller calculating diversions when a train reports that it has broken down or been stopped.

In the examples themselves, the outlines for such subsidiary algorithms are given largely in the form of comments, or as a mixture of comments and pseudo-code (both given in italics, but the former being enclosed in C-style comment delimiters, `/*...*/`, while parts of the latter may be enclosed in angle brackets, `<...>`).

5.1.2. Device Handling.

Code to perform device handling for external, hardware resources (sensors, actuators, etc.) may also be lengthy, particularly where sequences of torques, joint angles or other impulses need to be calculated. Also, the functions that provide the actual interface to the device will vary from system to system and from one type of device to another. For instance, in the case of an intelligent device controller, it might only be necessary to make an appropriate library call with the correct parameters, while other, less sophisticated, devices might have status or control registers mapped into the memory space or I/O vector of a processor, in which case appropriate values would need to be calculated and then assigned to the correct memory location or sent to the correct I/O port.

To illustrate this more clearly, consider the movement of a robot arm in the Quality Control example. If the robot arm is controlled by a sophisticated internal control sub-system, then the application level operation, `ARM_MOVE(Coords)`, may map directly to a system library call or the transmission of a message, containing appropriate parameters, to the arm controller's network address. Alternatively, if the controller interface for the robot arm is very basic and will only accept a list of joint angles and stepper motor impulses in some defined format, then all of the necessary calculations would need to be performed by the application and the final set of results sent to the appropriate controller address. This might entail a large set of intensive computations, mapping between different coordinate systems used by the application and the arm controller, working out appropriate angles and motor impulses for each individual joint in the arm and then converting all of these values into the appropriate controller format. The algorithm used would also have to be designed to avoid singularities in the arm's workspace (i.e. places which the arm cannot physically reach). It should immediately be apparent that the actual program code for such computations would probably be extremely long, even though the necessary mathematical transformations and control laws are, individually, very easy to express and program in any notation.

For the above reason, device handling code has largely been omitted from the examples. However, to show where such device handling operations are required, a high-level procedure call interface to the underlying hardware has been assumed. Such interface procedures are always given names in upper-case and any required parameters are passed to the operation in the normal way (e.g. `ARM_MOVE(Coords)`). It is assumed that such procedures simply provide a higher level interface to device controller operations such as those discussed above.

5.2. Call Control Example.

This application illustrates an event-driven system. A call controller object supports a dedicated bi-directional "hot-line" service between two telephones. When the receiver is lifted at one telephone, the other telephone begins to ring. If the ringing telephone is answered within a certain time then the call is connected, otherwise the call is abandoned, the ringing stops and a signal is transmitted to the calling telephone to alert the user to replace the receiver. Once a connection has been established, the call is not cleared until one party replaces their receiver (or both). The full sequence of events for a single call is therefore as follows:

1. Receiver lifted at caller's telephone, initiating call.

2. Callee's telephone rings for a pre-determined length of time.

3. If the receiver is lifted at the callee's telephone while it is still ringing, the ringing stops and the call is connected by the call controller. Alternatively, if the call is not answered within the pre-determined "ring time", the call controller will stop the callee's telephone ringing and transmit a "Call Disconnected" signal to the caller's telephone to alert the caller to replace their receiver.

4. If the call was connected successfully, it is allowed to continue until such time as either the caller or the callee hangs up. If only one side of the call is cleared in this way, then the call disconnected signal is, once again, sent to the other telephone.

This serves to define a *normal* call, however there are two other possible call sequences that might occur. In one, the caller gives up and replaces their receiver just as the callee answers, while in the other, the call is timed out and disconnected by the controller just as the callee answers. In the former case, the call controller must transmit the call disconnected signal to the callee's telephone, while the latter situation requires that the call disconnected signal be sent to both telephones.

The implementation of the system shown here is broken down into three objects of two distinct types: two identical telephone objects and a call control object. The telephone objects are, essentially, simple state machines, while the controller is an active object that receives invocations (signalling events) from the telephones and processes them according to the current state of the system (using the **select...accept** mechanism).

5.2.1. Telephones.

Two identical telephone objects provide software support for the telephones themselves. A telephone object offers an interface consisting of the operations (methods) `Ring` and `Stop_Ring`, to the call controller. These methods allow the controller to initiate or terminate ringing at a telephone. The latter operation also returns status information to the call controller, allowing it to deduce the current state of the telephone. Each telephone object also implements two private methods that are assumed to be invoked internally in response to appropriate signals from the telephone hardware. The `Receiver_Lifted` operation initiates or answers a call, depending upon the current state of the system, while the `Receiver_Dropped` operation terminates calls. From the point of view of the telephone object, a call takes place in the interval between a `Receiver_Lifted` invocation and the following `Receiver_Dropped` invocation.

```
INTERFACE Telephone IS
     Ring ;
     Stop_Ring returns Call_Status ;
END.

OBJECT Telephone IS

   STATE
     TelephoneNo MyNo ;
     Call_Status Current_State ;

   METHOD Receiver_Lifted
   action
     if Current_State = Call_Coming_In then
        begin
          STOP_RING ;
          ^Call_Control.Answer_Call ;     /* Asynchronous  */
        end                               /*   invocation. */
        else
            ^Call_Control.Start_Call(MyNo) ;
        Current_State := Call_in_Progress ;
    end action ;

    METHOD Receiver_Dropped
    action
        ^Call_Control.Clear_Call(MyNo) ;
        Current_State := No_Call ;
    end action ;

    METHOD Ring
    action
        START_RING ;
        Current_State := Call_Coming_In ;
    end action ;

    METHOD Stop_Ring RETURNS Call_Status
    action
        if Current_State = Call_Coming_In then
            begin
                STOP_RING ;
                Current_State := No_Call ;
            end ;
        return Current_State ;
    end action ;

END.
```

5.2.2. Call Controller.

The call controller software is an active object, offering methods for starting, answering and clearing calls. It is responsible for managing the connection between the telephones when a call is made or subsequently terminated. The internal thread of the call controller object uses an accept statement to wait for an incoming `Start_Call` invocation. Having received the `Start_Call` message, the controller waits for the call to be answered, abandoned or timed out (in the event of no-one answering) using the select mechanism. If the call is answered, the call controller waits for one party (or both) to clear the call. If only one party clears the line, the controller connects the remaining telephone to a "Disconnected" signal. This will also be done if a call is abandoned just as the callee answers or if the controller itself times out the call just as it is answered.

```
INTERFACE Call_Control IS
    Start_Call( IN TelephoneNo ) ;
    Answer_Call ;
    Clear_Call( IN TelephoneNo ) ;
END.
```

```
OBJECT Call_Controller IS

   STATE
     Telephone Caller, Callee ;      /* Hold caller and callee */
                                     /* identifiers while call */
                                     /* is in progress.        */
     Directory Phones ;     /* Contains mapping from telephone */
                            /* numbers to telephone object     */
                            /* identifiers.                    */

   METHOD Start_Call( IN TelephoneNo CallerNo )
   action
     CalleeNo := <Function of CallerNo - only 2 telephones.> ;
     Caller := Phones[CallerNo] ;
     Callee := Phones[CalleeNo] ;
     Callee.Ring
   end action ;

   METHOD Answer_Call
   action
     CONNECT(Caller, Callee)
   end action ;

   METHOD Clear_Call( IN TelephoneNo T )
   action
     DISCONNECT(Phones[T]) ;
     if Phones[T] = Caller then
       Caller := No_Connection ;
     else
       Callee := No_Connection
   end action ;

   THREAD
   cycle
     Caller := No_Connection ; Callee := No_Connection ;
     accept Start_Call ;
     select
       accept Answer_Call →
             accept Clear_Call →
                   begin
                     Disc := <Whoever didn't hang up>
                     Disconnection(Disc) ;
                   end ;
     □
       accept Clear_Call → if <Callee still ringing> then
                                Disconnection(Callee) ;
     □
       delay(Ring_Time) →  par
                             Disconnection(Caller) ;
                           ||
                             Disconnection(Callee) ;
                           end par ;
     end select ;
   end cycle ;
```

```
    PROCEDURE Disconnection( IN Telephone T )
    begin
      CONNECT(T, Disconnected_Signal) ;
      select
        accept Clear_Call ;
      □
        delay(Disc_Time) → DISCONNECT(T) ;
      end select ;
    end ;

  END.
```

5.2.3. Call Control Example : General Notes.

The major important feature of this call control example is that it shows a fully event-driven application. The use of the **select** and **accept** constructs in an active controller object, along with the use of asynchronous invocations, allows the software to respond to environmental events as and when they occur. The telephone objects need to use asynchronous invocations for communication with the controller because a telephone object must continue to monitor its local state while invocations are processed. For example, the receiver may be lifted, causing the telephone object to invoke the `Start_Call` operation, however while this invocation is being processed, the person who lifted the receiver might change their mind and terminate the call. In such cases, the telephone object must be able to detect that the receiver has been replaced and invoke the `Clear_Call` operation, even if the `Start_Call` operation is still being handled at the call controller. Hence, considering that invocations such as `Start_Call` or `Clear_Call` do not need to return values, it is appropriate to have telephone objects use asynchronous invocations for such events and, thereby, be able to continue monitoring their local state while such invocations proceed.

The call control application used in this example is relatively simple, however it is representative of a general class of event-driven applications: specifically, those applications where asynchronous activity in a system's environment triggers defined control operations. It is likely that the approach adopted in this example would be suitable for a wide range of applications of this type.

5.3. Assembly Line Quality Controller.

In this example, the system under consideration is an assembly line in a plant where containers of chemicals are processed (see [Davidson 89][Davidson 91][Wolfe 90]). Occasionally, a container may be defective, in which case it must be carefully removed and discarded, preferably without stopping the line. This task is carried out by two robot arms which also serve the line in other capacities. Defective containers are detected by a Quality Control system which then co-ordinates its activity with the two arms in order to lift the container from the line. In order to allow a faulty container to reach the arms, the lifting operation cannot begin until 5 seconds after detection. It must then be completed within 10 seconds of detection to make way for the next container to arrive. Hence, before a faulty container can be lifted, each arm must know that the current operating conditions will allow it to lift the container within the specified timing constraint. If the container cannot be grasped correctly or cannot be lifted, then the assembly line must be safely stopped, the container removed manually and the line reset. Similarly, if the deadline expires while the arms are still in the process of lifting their load, the assembly line must again be stopped and the arms cleared by an operator in order to prevent spillage or other hazards.

This application is a classical process control problem where independent controllers must co-ordinate their activity in order to perform some task. As with the call control system, the implementation is broken down into three objects of two distinct types (two arm controllers and a quality monitor).

5.3.1. Quality Monitor.

This is an active object that continuously monitors containers as they pass down the line. When a faulty container is detected, a local `Remove_Container` operation is invoked. This first of all calls parallel `Prepare_Lift` operations at the two arm controllers. If both of these preparatory operations succeed then two parallel `Perform_Lift` operations are invoked, otherwise the line is stopped and reset. Expiry of the deadline for removing the container during either the `Prepare_Lift` or `Perform_Lift` phases causes the quality monitor to shut down the assembly line.

```
OBJECT Quality_Monitor IS

  STATE
    Arm_Controller Arms[2] ;
    Status Line_Status ;

  COMMIT
    DEFAULT : <State-based commit> ;

  ABORT
    HALT : /* Stop assembly line and alert   */
           /* operator to remove container. */

  METHOD Remove_Container( IN Time Start_Time, IN Time My_DL )
  action
    integer i ;
    Status_Flag C_Flag ;

    do
      par for i := 1 to 2
        Arms[i].Prepare_Lift(My_DL) [ Fail : Abort(HALT) ] ;
      end par ;

      par for i := 1 to 2
        Arms[i].Perform_Lift(My_DL) [ Fail : Abort(HALT) ] ;
      end par
    start_in (Start_Time) ;
    finish_in (My_DL) [ Deadline : Abort(HALT) ] ;

  end action ;

  THREAD
    begin
    every Period do
      if <Current container defective> then
        Remove_Container(5s,10s) ;
    end ;

END.
```

5.3.2. Arm Controllers.

The robot arms that co-operate to remove defective containers from the line also serve the line in other capacities. The object that corresponds to the arm controller will therefore have a range of operations relating to its normal activities. In the interests of clarity and brevity, these operations are not explicitly shown in the following implementation, since the only operations that are of interest for this example are the `Prepare_Lift` and `Perform_Lift` operations. The former attempts to move the arm into position ready to pick up the faulty container from the line. If the movement fails (for example because the arm cannot reach the appropriate position) or the timing constraint is violated, the arm is locked in position and a `Fail` exception is raised to the quality monitor to initiate further recovery. Alternatively, if the `Prepare_Lift` operation is successful, the arm controller will then wait for the corresponding `Perform_Lift` operation to be invoked. This causes the arm to grasp the faulty container and attempt to remove it. Any failure during this activity again causes the arm to be locked in position and a `Fail` exception raised.

To ensure that the arm controllers perform their operations in the correct sequence, specifically only allowing `Perform_Lift` to follow `Prepare_Lift`, the arm controller is also an active object in which the internal thread uses `select` and `accept` to guarantee the correct sequencing of these operations.

```
INTERFACE Arm_Controller IS
    Emergency_Release ;
    Prepare_Lift( IN Time ) ;
    Perform_Lift( IN Time ) ;
END.

OBJECT Arm IS

  STATE
    Position Current ;

  COMMIT
    DEFAULT : <State-based commit> ;
  ABORT
    LOCK : <Lock arm in position>

  METHOD Prepare_Lift( IN Time My_DL )
  action
    do
      /* Work out motion plan to reach pick-up point. */
      ARM_MOVE(<Pick up point>) ;
    finish_in (My_DL) [ Fail, Deadline :  begin
                                            Abort(LOCK) ;
                                            signal Fail ;
                                          end ] ;
  end action ;

  METHOD Perform_Lift( IN Time My_DL )
  action
    do
      ARM_GRAB ;
      /* Work out motion plan to raise arm. */
```

```
            ARM_MOVE(<Motion plan>) ;
         finish_in (My_DL) [ Fail , Deadline : begin
                                                    Abort(LOCK) ;
                                                    signal Fail
                                                 end ] ;
      end action ;

      THREAD
         cycle
            select
               accept Prepare_Lift -> accept Perform_Lift ;
            □
                  /* Other robot arm services.*/
            end select ;
         end cycle ;
   END.
```

5.3.3. Assembly Line Example : General Notes.

This example has a number of features that are common to a range of process control applications. The requirement for programmed error recovery, the co-ordination of several different sub-system controllers to perform a task, timing constraints on operations and a cyclic monitor that initiates a defined sequence of activity when it detects a certain event in the system's environment (the arrival of a faulty container in this case). From the point of view of the object model and programming notation proposed here, all of these features can be easily supported and expressed. The availability of programmed commit and abort operations and an exception handling mechanism makes programmed recovery straightforward while the constructs proposed for timing constraints make it easy to specify start times and deadlines for operations. Since objects can be active, a cyclic monitor object can be programmed with little difficulty and the use of the select and accept constructs once again (as in the previous example) allows the activity of the system to be limited to a defined execution path when required.

5.4. The A.S.V. (Autonomous Suspension Vehicle).

The Autonomous Suspension Vehicle (ASV) is a six-legged walking robot that moves by taking a sequence of steps determined by some overall motion plan (see [Gheith 89][Gheith 90]). Its six legs are arranged in three pairs (front, centre and rear) and each leg is controlled by a separate actuator. A single step consists of moving each leg forward individually and then pushing all of the legs backwards simultaneously, hence moving the body of the ASV forwards. To maintain stability, certain constraints are placed upon the movement of the legs. First of all, a corner leg (any one of the front or rear pairs) can only be moved when all of the other legs are safely on the ground. The middle pair of legs however can both be moved together, so long as all four corner legs are safely in place. If the stability of the ASV is threatened by the violation of any of these constraints then recovery actions must be taken such as forcibly placing a failed leg back in contact with the ground. Similarly, if stability is threatened by an unbalanced load, then all legs may have to be placed on the ground. Finally, there is also a requirement that each leg does not remain off the ground for longer than a specified time, τ, while it is moving. This is to prevent excess strain on the body of the robot.

Like the previous example, this system illustrates a general class of plant control applications, however the structuring of the control software in this case is much more hierarchical in nature. Three distinct object types are required, one for the central ASV controller, one for the leg controller software and one to provide an appropriate synchronisation mechanism so that the constraints on the movement of legs are not violated. For the sake of brevity, this third object will not be shown and the following outline only gives the implementation of the ASV and leg controllers. (Note that the synchronisation object is only needed because this example attempts to match as closely as possible the functionality of the ASV as described in [Gheith 90]. Given an alternative implementation of the system, in which the individual legs of the ASV are placed more directly under the control of the master ASV controller, the need for a synchronisation object could be avoided.)

5.4.1. Master Controller.

The ASV controller object offers an interface which includes the operation `Move`, possibly along with other status or control operations not shown here. It is assumed that calls to the ASV controller are issued by some component in the user interface for the ASV and that exceptions, where raised, are returned to the user-interface level. Returning to the `Move` operation, it takes a position and a deadline as its parameters and it attempts to move the robot to the specified position within the given time. A move is performed as a sequence of `step` invocations, each intended to move the robot nearer to its final destination. A single step operation consists of making appropriate invocations to the leg controllers to move the legs and then push all of the legs backwards. Failures during any of these operations may cause the ASV controller to initiate appropriate recovery, either stopping the ASV entirely or adapting its activity to match the current state of the application (for example, re-trying a failed `step` operation or re-calculating the overall motion plan).

The master ASV controller is also an active object, containing a background thread that monitors the robot's stability. In the event of the ASV becoming unstable, this background thread forces the ASV to shut down and alerts the operator. A similar emergency procedure is followed if the robot becomes unstable due to a leg failure.

```
INTERFACE ASV_Control IS
     Move( IN Position, IN Time )
        SIGNALS Danger, Out_of_Time ;
END.

OBJECT ASV IS

   STATE
      Position Current ;
      Leg_Controller Legs[6] ;

   COMMIT
      DEFAULT : <State based commit> ;

   ABORT
      DEFAULT : <State restoration> ;
      STEP_FAILED : /* Application specific recovery. */
      PANIC :   begin
                   HALT_ASV ;.
                   /* Set off alarm to alert operator. */
                end ;

   METHOD Step( IN Position dest, IN Time My_Deadline)
      SIGNALS Danger
   action
      Position LegPos[6] ;
      integer i ;
      Time Push_Time ;

         do
         par for i := 1 to 6
            LegPos[i] := <New position for leg i>
        Legs[i].Move(LegPos[i])
                    [ Fail: <Record move failure for leg i> ] ;
      end par ;
      if <Any move failed> then Abort(STEP_FAILED) ;

      Push_Time := <NOW + δ > ;
      par for i := 1 to 6
        Legs[i].Push(Push_Time)
                    [ Fail: <Record push failure for leg i> ;
                      Push_Failure: signal Danger ; ] ;
      end par ;
      if <Any push failed> then Abort(STEP_FAILED) ;
    finish_in (My_Deadline)
                    [ Deadline : Abort(STEP_FAILED) ; ] ;
  end action ;

  METHOD Move( IN Position Dest, IN Time Master_DL )
    SIGNALS Danger, Out_of_Time
  action
    Time i_deadline ;       /* Intermediate step deadline.    */
    Position i_pos ;        /* Intermediate step positions.   */
    boolean Step_OK ;

    do
      while Current != Dest do
      begin
        i_deadline := <Function of Master_DL and time> ;
        i_pos := <Function of Current and Dest> ;
        Step_OK := TRUE ;
        do
          Step(i_pos, i_deadline)
                    [ Fail : Step_OK := FALSE ;
                      Danger : if <ASV unstable> then
                                 begin
                                   Abort(PANIC)
                                   signal Danger ;
                                 end
                               else
                                 Step_OK := FALSE ; ] ;
        finish_in i_deadline
                    [ Deadline : Step_OK := FALSE ; ] ;
        if Step_OK then
          Current := i_pos
        else
          /* Calculate new current position. */
    finish_in Master_DL
                 [ Deadline : signal Out_of_Time ; ] ;
  end action ;

  THREAD
  begin
    every 5s do
      if <ASV unstable> then Abort(PANIC) ;
  end ;

END.
```

5.4.2. Leg Controllers.

The master ASV controller relies upon six individual leg controllers to handle the movement of the robot's legs. Each leg controller provides a `Move` and a `Push` operation. The former takes as a parameter a new position to which the leg must be moved and, obtaining permission from the leg synchronisation object (`Move_Permission`), attempts to move the leg to the new position within τ time units. If the movement does not succeed or the leg is not returned to the ground in time, a recovery operation is called to put the leg back on the ground and the default failure exception will be raised to the master controller. The `Push` operation takes as a parameter a start time at which the leg should be pushed backwards and it attempts to perform the necessary push at the appropriate time. If the operation fails, it will either raise the leg from the ground causing the default failure exception to be signalled (if this is the first failure), or it will explicitly signal a `Push_Failure` exception, causing the master controller to halt the ASV (since its stability may now be jeopardised).

```
INTERFACE Leg_Controller IS
     Move( IN Position ) ;
     Push( IN Time )
        SIGNALS Push_Failure ;
END.

OBJECT Leg IS

   STATE
     Leg_Type My_Type ;        /* = MIDDLE or CORNER.  */
     Position Current ;

   ABORT
     PUSH : RAISE_LEG ;
     MOVE : begin
               if <Leg clear of ground> then
                  LOWER_LEG ;
               Current := <Final position of leg> ;
               Move_Permission.Release(My_Type) ;
            end ;

   METHOD Move( IN Position Dest )
   action
     if Move_Permission.Request(My_Type) != GRANTED then
          Abort ;
     /* Now, move leg from current position to destination.*/
     /* Must perform move within τ  time units to minimise    */
     /* stresses on robot body.                             */
```

```
      do
         MOVE_LEG(Dest) ;
      finish_in (τ) [ Fail, Deadline : begin
                                          Abort(MOVE) ;
                                          signal Fail ;
                                       end ] ;
      Move_Permission.Release(My_Type) ;
   end action ;

   METHOD Push( IN Time Start_Time )
      SIGNALS Push_Failure
   action
      /* At specified time, push leg backwards. */
      do
         PUSH_LEG [ Fail :  if <This is first failure> then
                               begin
                                  Abort(PUSH) ;
                                  signal Fail ;
                               end
                            else
                               signal Push_Failure ] ;
      start_in (Start_Time) ;
   end action ;

END.
```

5.4.3. ASV Example : General Notes.

The ASV control system falls into the same general class of applications as the assembly line controller shown earlier. However, where the quality monitor and the arm controllers of the previous example are independent sub-systems that occasionally co-operate to perform a certain task, the interaction of the control sub-systems in this example is much more hierarchical, with the leg controllers being subordinate to the master ASV controller at all times. This hierarchical arrangement is easily expressed in the object model by having the leg controllers declared as part of the internal state of the master controller. The most important features of the object model for this particular application are probably the availability of programmed recovery operations, exception handling and timing constructs. These three mechanisms are central to much of the activity of both the master controller and the leg controllers: the former because it must attempt to meet the deadline imposed upon the move operation and it must be able to adapt its motion plan to changing circumstances; the latter because they must interface with the leg control hardware and perform compensating operations when legs fail. Being able to make the master controller an active object and include a background thread to monitor the stability of the ASV is also a useful feature.

5.5. Train Control Example.

This example gives an outline for the implementation of the most sophisticated system yet considered for this object model. The application is a reliable real-time control system for a large, digitally-controlled model train layout. The amount of information which must be stored and processed is much greater than that in previous examples, since a map of the overall track layout is required (including connectivity information), a timetable (with route information) must be maintained and appropriate control and state information must be managed for each individual train and each section of track. In order to simplify the presentation of the controller objects, it will be assumed that a number of user-defined object types are already available, supporting an appropriate set of data structures for the representation of such information. For reference, these are briefly described below:

TrackList

A TrackList is a linked list containing IDs of track sections. Instances can be used to hold an overall list of the track sections in the layout and to hold low-level route information. (Individual elements of the list are instances of the class TrackListEntry.)

StageList

This is another linked list containing route information for train journeys. It consists of a number of TrackLists, each representing one stage of a train's journey. Each stage also has a stage number and an associated start time and deadline. (Individual list elements are instances of the class StageListEntry.)

TTEntry

A TTEntry consists of a StageList (which represents a train journey) along with a Train ID (the train which will make that journey) and some status information (journey in progress, journey completed, journey abandoned, etc.).

TimeTable

A TimeTable is simply a structured collection of TTEntries. (It would probably be implemented as yet another linked list.)

TrainPos

This is an implementation of a simple data record which can be used to hold position information (i.e. journey stage number and current track section number) for trains.

D_List

This is an array or a list of TrackLists, each of which holds the route information required to divert trains around a known breakdown. The D_List can be indexed by the section IDs of the blocked sections of track and can be allowed to grow and shrink dynamically as trains break down and are subsequently cleared.

C_Map

An array structure to hold connectivity information for the track layout.

P_Map

Instances of this class hold mapping information for sets of points. (Any track section which includes a set of points will have a points map relating points settings to track section IDs - the points can then be set correctly to pass trains on to the appropriate next section on their journey.)

The application software is broken down into a number of control objects. A central controller manages information regarding the layout of the track, routes and the timetable for train journeys. This controller interacts with a number of individual train controllers (one for each train), instructing them to begin their journeys, giving them route or timing information and providing assistance when problems arise (i.e. working out diversions, etc.). There are also a number of track objects (one for each section of track). These handle reservations, as and when requested by the train controllers and provide route control by setting points appropriately.

5.5.1. Layout Controller.

This is the central controller part of the system and its main function is to act as an information base and co-ordinator for all of the trains. It holds a list of all the track sections in the layout, as well as connectivity data for the sections. It reads a timetable from some source (e.g. a file on disk) and trains are then run along specified routes at specified times in accordance with the timetable entries. When a train completes a journey successfully, its timetable entry is logged as completed. However, it is possible that some trains may encounter problems during their journey, so appropriate recovery actions are also supported by the layout controller. For example, a train may break down. This causes the controller to log the broken train's timetable entry as failed and alert an operator to clear the occupied sections of track. The controller also works out an alternative route (or routes) that can be taken by trains that are held up as a result of the breakdown.

```
INTERFACE Controller IS
     Load_Timetable( IN Timetable ) ;
     Re_Route( INOUT TTEntry, IN TrainLocation ) ;
     Re_Schedule( INOUT TTEntry, IN TrainLocation ) ;
     Start_Trains ;
END.

OBJECT Train_Controller IS

   STATE
     TrackList Sections ;
     C_Map Connectivity ;
     TimeTable TT ;

   COMMIT
     COMPLETE :  begin
                   TTE.Status := COMPLETED ;
                   exit ;
                 end ;
   ABORT
     FAILED :  begin
                 TTE.Status := FAILED ;
                 exit ;
               end ;
```

```
    METHOD Clear_Line( INOUT TTEntry TTE, IN Train ThisTr )
    action
      /* Find location of train and alert operator to clear */
      /* line. Also work out a diversion which other trains */
      /* can use to avoid the blocked sections of track.    */
    end action ;

    METHOD Run_Train( INOUT TTEntry TTE )
    action
      Train ThisTrain  ;

      ThisTrain := get_server(TTE.TrainID)
                                 [ Fail : Abort(FAILED) ] ;
      ThisTrain.Set_Route(TTE) [ Fail : Abort(FAILED) ] ;
      ThisTrain.Run [ Fail   : Abort(FAILED) ;
                      Broken : begin
                                 Clear_Line(TTE, ThisTrain) ;
                                 Abort(FAILED)
                               end ; ]
    end action ;

    METHOD Re_Route( INOUT TTEntry TTE, IN TrainLocation TL )
    action
      /* Work out an alternative route for this train or  */
      /* force the train to wait.                         */
    end action ;

    METHOD Re_Schedule( INOUT TTEntry TTE, IN TrainLocation TL )
    action
      /* Work out a new journey schedule for this train.  */
    end action ;

    METHOD Load_Timetable( IN Timetable T )
    action
      TT := T
    end action ;

    METHOD Start_Trains
    action
      integer i ;

      par for i := 1 to TT.No_of_Entries
        Run_Train(TT.Entry(i)) ;
      end par
    end action ;

  END.
```

5.5.2. Train Controllers.

Each train has a corresponding control object in the application software. Each train controller has an identifier, a route, a speed and a current position. After receiving its instructions (route, etc.) from the layout controller, the train controller handles the train on its journey along the assigned route, reserving and releasing sections of track as it proceeds. If, for some reason, a train cannot begin its journey, a default failure exception will be signalled to the layout controller. Similarly, if the train breaks down while en route, the Broken exception is raised and the layout controller will then provide appropriate error recovery.

Another possible problem is that a train may be unable to reserve a section of track because some other train is already using it. In this case, a request is made for an alternative route and the layout controller will either supply diversion information or force the train to wait. Finally, each train controller includes an internal thread that monitors the train's progress (according to its timetable). If the train is running late and the delay is outside some acceptable bound, a request will be made to the layout controller to have the train re-scheduled.

```
INTERFACE Train IS
    Set_Route( IN TTEntry TTE ) ;
    Locate returns TrainLocation ;
    Run
      SIGNALS Broken ;
END.

OBJECT Train IS

  STATE
    integer TrainID, Speed ;
    TTEntry MyRoute ;
    TrainLocation CurrentPosition ;
    Time Monitor_Period ;
    boolean Running := FALSE ;

  COMMIT
    STOP : begin
             HALT ;
             Running := FALSE
           end ;
    NULL : skip ;

  ABORT
    STOP : HALT ;
    NULL : skip ;

  METHOD SetRoute( IN TTEntry TTE )
  action
    MyRoute := TTE ;
    if TTE.Route.FirstSection.Acquire != GRANTED then
      Abort ;
  end action ;
```

```
METHOD Locate RETURNS TrainLocation
action
  return CurrentPosition
end action ;

METHOD MoveCycle( INOUT StageList S )
  SIGNALS Breakdown, Blocked
action
  integer i ;

  for i := 1 to <Number of sections remaining> do
  begin
    Speed := <Required speed to complete stage on time> ;
    SET_SPEED(Speed) [ FAIL : begin
                                Abort ;
                                signal Breakdown ;
                              end ; ] ;
    do
      <Attempt to acquire next section of track.>
    finish_in (<Time to next section>)
                    [ Deadline :  begin
                                    Abort ;
                                    signal Blocked
                                  end ; ] ;

    Next_Section.Set_Points(Curr_Section, Dest_Section) ;

    /* Wait until train has cleared previous section  */
    Last_Section.Release() ;

    /* Update CurrentPosition, route information,     */
    /* server information etc.                        */
  end ;
end action ;

METHOD Run  SIGNALS Broken
action
  integer i ;

  Running := TRUE ;
  for i := 1 to MyRoute.Route_Info.No_of_Stages do
  begin
    do
      while <Stage incomplete> do
        MoveCycle(Current_Stage)
            [ Fail : if <This is the first failure> then
                         <Make another attempt>
                     else
                         Abort ;
              Blocked : Controller.Re_Route
                                (MyRoute,CurrentPosition) ;
              Breakdown : signal Broken ;
            ] ;
    start_in (Current_Stage.Start_Time) ;
    Current_Stage := Current_Stage.Next ;
  end ;
end action ;
```

```
      THREAD
         every Monitor_Period do
            if Running and <Train running late> then
               Controller.Re_Schedule(MyRoute, CurrentPosition) ;

   END.
```

5.5.3. Track Controllers.

There is a track controller object for each section of track in the layout. Each section has a unique identifier, a list of adjacent track sections and a points map which gives the appropriate points setting for routing trains to adjacent sections. (Naturally, straight sections without points have empty points maps.) Each section controller also contains a flag to indicate whether the section is currently reserved, the identifier of any train currently holding the section, a record of the current points setting and a default points setting which is used to set the points when the system is initialised.

When trains are running, a track section will receive a stream of reservation requests from different trains. For a given request, if the section is currently available for that train, the request is granted and the appropriate reservation made. If the section is not available, the request fails and appropriate recovery is left in the hands of the train and layout controllers. For a train that has reserved a section, the train controller can request that the points be set appropriately. For this operation, the train controller specifies the next section that it needs to reach and the track controller selects the correct points setting from the points map.

```
INTERFACE Track IS
     Acquire( IN integer ) RETURNS Lock_Status ;
     Set_Points( IN integer, IN integer ) ;
     Release( IN integer ).;
END.

OBJECT Track IS

   STATE
     integer TrackID, TrainID, Default_Points_Setting ;
     Connections Adjacent_Tracks[MAX_TRACK_CONNECTIONS] ;
     P_Map Points ;
     boolean Locked ;

   METHOD Set_Points( IN integer IncomingID,
                      IN integer DestinationID )
   action
     /* Look up appropriate points setting to route the  */
```

```
      /* incoming train to the right destination.         */
      SET_POINTS(<Appropriate Setting>)
    end action ;

    METHOD Acquire( IN integer TID ) RETURNS Lock_Status
    action
      if not Locked then
        begin
          Locked := TRUE ;
          TrainID := TID ;
          return GRANTED ;
        end ;
      else
        return FAILED ;
    end action ;

    METHOD Release( IN integer TID )
    action
      if <Locked by train TID> then
        begin
          Locked := FALSE ;
          TrainID := 0 ;
        end ;
    end action ;

  END.
```

5.5.4. Train Control Example : General Notes.

This example is interesting since it potentially represents a larger scale of application than the previous examples and it is a hybrid system in that it consists of process control elements (notably the train controllers) and a real-time information processing component (the layout controller and, to a certain extent, the track controllers). The pattern of control interactions in the system is also a hybrid, having both hierarchical and co-operative elements. For example, interactions between the layout controller and the train controllers are, basically, hierarchical with the train controllers being subordinate to the layout controller whereas interactions between train controllers and track controllers are of a more co-operative nature with neither object strictly having control over the other.

From the point of view of the object model, this example serves three major purposes. Firstly, it illustrates that the object model and notation can be used to express a very general class of real-time information processing applications. Secondly, it highlights the potential for easily combining process control sub-systems that include operation-based recovery with information processing sub-systems that use state-based recovery. Finally, it shows that a complex pattern of activity and interactions between a large number of objects can still be supported within the framework imposed by the object model.

5.6. Chapter Summary.

This chapter has demonstrated the versatility of the object model and programming notation that was described in chapter 4. This has been done by giving brief outline implementations of four different examples, ranging in complexity from a relatively simple call-control system to a more complex real-time control system for a model train layout. Each example also included a brief discussion of the general class of application that it represented and listed the two or three specific features of the object model that are likely to be most useful for applications of that type. Throughout all of the examples, none of the constraints imposed by the object model have been violated, yet a wide range of different activities and system behaviours have been supported. This can be taken as a straightforward, intuitive argument both for the general expressivity of the proposed programming notation and for the useful potential of the object model upon which it is based.

Chapter 6.

Architectural Support for Real-Time Objects.

In the previous chapter, the versatility of the real-time object model proposed in chapter four was illustrated by means of four different application examples. However, one important aspect of the object model that has not yet been discussed is its implementation, either in terms of specific mechanisms (e.g. the atomic action mechanism) or in terms of the underlying architectural support that would be required to support object replica groups constructed using this model. Such issues are important, insofar as the object model is intended to be geared towards the programming of real-time applications and, as such, must itself be capable of being implemented.

Unfortunately, any attempt to describe a complete implementation of all of the features and mechanisms included in the model would be both long and complex. The purpose of this chapter is, therefore, to give a brief outline of the way in which some of the more important parts of the object model can be supported. This will include an overview of the support required for general features of the model, such as the exception handling and timing constraint mechanisms, as well as the method selection (`select...accept`) primitive. The provision of application specific recovery operations within the atomic action mechanism will then be discussed. Finally, three specific multicast protocols will be considered to show the way in which object replica groups can be supported: in one case, on fail arbitrary processing nodes and, in the others, on nodes that are fail-silent.

6.1. General Features.

Although the object model does include certain features that are not common in current programming languages, much of the general structure of the model is based upon existing languages or systems. It is based upon the imperative style of programming, like many common languages such as Pascal, Modula 2 or C, and the overall structure of objects and their interfaces is broadly similar to that found in existing object-oriented languages such as C++. The concept of active objects is not new, having already been used in systems such as ARTS and CHAOS, and the use of both synchronous and asynchronous invocation primitives reflects a similar approach to communication found in the CONIC system [Kramer 83][Sloman 87]. The fact that such similar mechanisms exist in other systems and languages means that the implementation of these aspects of the object model is unlikely to present any new problems and need not be discussed in any great detail.

Given that the general structure of the model is implementable, there are specific mechanisms that remain to be considered. In particular, the atomic action and exception handling mechanisms have not been examined and the temporal scope and method selection mechanisms are not wholly based upon existing constructs. The implementation of forward recovery within atomic actions will be described in more detail in section 6.2, but first, the exception handling, temporal scope and method selection constructs must be discussed.

6.1.1. Exception Handling.

The exception handling mechanism that has been used in the object model is based upon the termination model of exception handling and, as such, it is similar to mechanisms used in existing languages like CLU. In fact, the semantics of the exception handling functions and the syntax used for exception handling in the examples have been developed directly from the mechanisms described in [Cristian 82] and [Cristian 89] and no real changes have been made to their basic functionality. Hence, the implementation of such an exception handling mechanism, like the implementation of the object model's general programming constructs, should prove to be relatively straightforward and could be based upon an existing implementation used in some other system.

6.1.2. Timing Constraints.

The temporal scope constructs that have been used to express timing constraints are similar to the language constructs used to express timing constraints in MPL (the MARUTI Programming Language), however there are two major differences that should be noted.

First of all, MPL uses six basic timing primitives, whereas the object model proposed here uses only two, more general constructs. From the point of view of an underlying real-time scheduling mechanism, there is no real difference between these two approaches and the mapping between application-level timing constraints and task scheduling parameters could be implemented in a similar manner in both cases. The other, more fundamental, difference between the constructs used here and those used in MPL is that the MPL timing primitives do not include the use of exceptions or exception handling to deal with timing errors. This is because the MARUTI system employs pre-scheduling for all real-time tasks and it can, therefore, be assumed that tasks will not exhibit timing errors at run-time. However, the integration of the exception handling mechanism that has already been included in the object model with the temporal scope mechanisms would be unlikely to be difficult.

6.1.2.1. Replicated Timing Constraints.

The implementation of timing constraints across a replica group is another issue that must be considered. If each replica in a replica group is executing a temporal scope, differences in clock synchronisation or communications delay between different replicas can lead to situations where some replicas regard their timing constraint as being met while others regard it as being missed. This can lead to state divergence between different replicas, so some mechanism is required whereby decisions regarding timing constraints can be agreed across the replica group. In chapter three, it was shown that a temporal scope of the form:

```
do
  S ;
start_in(EST,LST) ;  [ Early_start : A ; Late_start : B ] ;
finish_in(EFT,LFT) ; [ Early_finish : X ; Late_finish : Y ] ;
```

permits an equivalent implementation, using the **select...accept** mechanism, that will allow events such as `Early_start`, `Late_finish` or the successful completion of `S` to be agreed between replicas. The correct implementation of replicated timing constraints therefore depends upon an implementation of the **select** mechanism that is deterministic across replica groups. Such a mechanism is already required within the object model and its implementation is described below.

6.1.3. Method Selection.

Considering method selection in active objects, a different implementation problem presents itself. The **select...accept** construct that can be used to control the execution of an active object is directly based upon a similar mechanism used to handle task entry calls in the Ada language. Its implementation should, therefore, pose no problems. This would be true if the object model did not permit the use of active replication, but the fact that an object may be part of an active replica group means that an alternative implementation of the **select...accept** mechanism must be developed which prevents replica state divergence by ensuring that non-deterministic selection can be agreed across replica groups.

```
procedure receivefrom(P:setof ports, t:timevalue)
     returns m:message
begin
     within t do
          { m := RECEIVEFROM(P) }      /* Returns first message   */
     timeout:                          /* in queue from Pi∈ P.    */
          { send(self, marker)
            m := RECEIVEFROM([P,self]) }
     od
     if m = marker → m := null
     □ m ≠ marker → skip
     fi
end
```

Figure 6.1. The generic input function.

Fortunately, such a "deterministic" implementation of the **select...accept** construct is possible and it has already been described in some detail in chapter three. At the most fundamental level, it relies upon the use of an atomic broadcast protocol or similar mechanism to ensure ordered message queues at each replica. Each replica then uses the generic input function (see figure 6.1) to select the first suitable message from its local queue. In those cases where an appropriate message has arrived from some source, this guarantees that each replica will select the same message. Similarly, the use of marker messages to signal timeouts ensures that such events are processed consistently across the replica group. It can then be guaranteed that either all of the replicas will accept and process the same message or all of the replicas will regard the input operation as having timed out.

6.2. Atomic Actions and Application Specific Recovery.

Several systems have been developed that use atomic actions to provide fault tolerance and the implementation of state-based atomic action mechanisms is quite well understood. However, the atomic action mechanism proposed in chapter 4 is slightly different in that it also allows application specific, forward recovery operations to be defined. Consideration must therefore be given to the way in which such an extended atomic action mechanism might be implemented. The approach suggested here is based upon the current implementation of atomic actions in the Arjuna system and it illustrates one way in which application specific recovery might be supported.

6.2.1. The Arjuna Atomic Action Mechanism.

The Arjuna system [Shrivastava 91b][Dixon 88] provides tools that assist in the construction of fault-tolerant, distributed applications structured as atomic actions operating on persistent objects. Arjuna is written in C++ and makes extensive use of the type

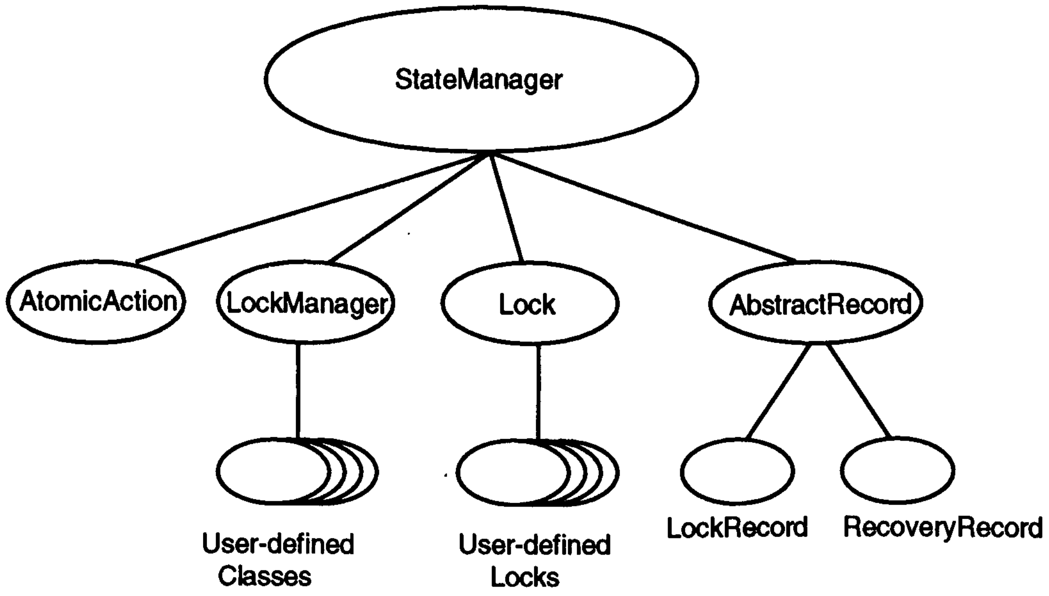


Figure 6.2. The Arjuna class hierarchy.

inheritance facilities provided by that language to allow user-defined objects to inherit desirable characteristics such as persistence or recoverability. The Arjuna class hierarchy is shown in figure 6.2.

The class StateManager provides an interface to the Arjuna object store, allowing instances of all classes derived from StateManager to be persistent. User-defined classes that are to be accessed using atomic actions are derived from the class LockManager, with the class Lock providing standard two-phase locking facilities. If any special forms of locking are required, this can be achieved by deriving new classes from Lock. The co-ordination of all these mechanisms to allow the application programmer to begin, end and abort atomic actions is provided by the class AtomicAction and it is this class which is of most interest here.

```
class AtomicAction : public StateManager
{
  ...
  protected:
    PrepareOutcome Prepare() ;
    void Commit() ;
  public:
    static AtomicAction *Current ;

    AtomicAction() ;
    AtomicAction(Uid*) ;
    virtual ~AtomicAction() ;

    virtual bool save_state(ObjectState*, object_type) ;
    virtual bool restore_state(ObjectState*, object_type) ;
    ...
    virtual Action_Status Begin() ;
    virtual Action_Status End() ;
    virtual Action_Status Abort() ;
    ...
    bool add(AbstractRecord*) ;
    AtomicAction* Parent() ;
} ;
```

Figure 6.3. The Arjuna AtomicAction *interface.*

Figure 6.3 shows part of the interface to the AtomicAction class. Of the operations shown, the two that are of most relevance to the current discussion are `End()`, which is called by the user to commit an action and `Abort()`, which is called to abort an atomic action. Outline descriptions of the current implementation of these operations are shown in figures 6.4 and 6.5.

```
ActionStatus AtomicAction::End()
{
  /* Check for superfluous invocation. */
  if ((actionStatus == COMMITTED) || (actionStatus == ABORTED))
    return actionStatus ;
```

```
        /* Ensure that this is the currently active action. */
        if ((currentAct != 0) && (currentAct != this) &&
              (currentAct->isAncestor(get_uid())))
        {
           /* Active action is one of this action's children. */

           /* Prevent Commit of parent actions (ensures safety). */
           /* Abort child actions. */
        }

        if (actionStatus == RUNNING)
        {
           if (prepare() == PREP_NOTOK)
           {
              /* Phase 1 of commit protocol (preparing to commit  */
              /*   state changes to stable storage) has failed    */
              /*   for some reason, so abort the action.          */
              phase2Abort() ;
           }
           else
           {
              /* Prepare phase completed successfully, so go ahead  */
              /*   and commit the action. (Commit state changes.)   */
              phase2Commit() ;

              /* For top-level actions or nested top-level actions  */
              /*   remove intention list from object store.         */
           }
        }

        return (actionStatus) ;
     }
```

Figure 6.4. Outline of `AtomicAction::End()` *operation.*

Using the existing operations as a starting point, it should be possible to implement an atomic action mechanism that permits application specific commit (`End()`) and abort operations as follows. First of all, allow each operation to take a parameter that specifies the type of commit or abort processing that is to be carried out. Then, within each operation, use a **switch** (**case**) statement on that parameter to select the appropriate operation, with the default selection being the state-based commit or abort shown above. Such an implementation of the `Abort()` operation is outlined in figure 6.6.

```
     ActionStatus AtomicAction::Abort ()
     {
        /* Check for superfluous invocation. */
        if ((actionStatus == COMMITTED) || (actionStatus == ABORTED))
           return actionStatus ;

        /* Ensure that this is the currently active action. */
        if ((currentAct != 0) && (currentAct != this) &&
             (currentAct->isAncestor(get_uid())))
        {
          /* Active action is one of this action's children. */

          /* Prevent Commit of parent actions (ensures safety). */
          /* Abort child actions. */
        }

        if ((actionStatus == RUNNING || actionStatus == PREPARING ||
             actionStatus == UNPREPARED ||
             actionStatus == PREPARED || actionStatus == ABORTING))
        {
          actionStatus = ABORTING ;

          /* Abandon any state changes that the action would have  */
          /* caused. (Information stored in pendingList).          */
          doAbort(*pendingList) ;

          actionStatus = ABORTED ;
          currentAct = parentAction ;
        }

        return (actionStatus);
      }
```

Figure 6.5. Outline of `AtomicAction::Abort()` *operation.*

This is, perhaps, the simplest way in which application specific commit and abort operations could be supported within the Arjuna framework and other implementations could be developed that would present a more efficient and friendly interface to the applications programmer. However, as far as the object model proposed in this thesis is concerned, the important point is that the implementation of an atomic action mechanism with application specific recovery and commit operations is, indeed, possible.

```
      ActionStatus AtomicAction::Abort ( AbortType AT )
      {
        /* Check for superfluous invocation. */
        if ((actionStatus == COMMITTED) || (actionStatus == ABORTED))
          return actionStatus ;

        /* Since all actions are top-level (or nested top-level)    */
        /*   there is no need to check if this is the currently     */
        /*   active action or to process parent or child actions.   */

        if ((actionStatus == RUNNING || actionStatus == PREPARING ||
             actionStatus == UNPREPARED ||
             actionStatus == PREPARED || actionStatus == ABORTING))
        {
          actionStatus = ABORTING ;
        switch(AT)
        {
          case TYPE_1:
                          /* User-defined abort operation 1.  */
                          break ;
          case TYPE_2:
                          /* User-defined abort operation 2.  */
                          break ;
          case TYPE_3:
                          /* ...and so on. */
                          break ;
          default:
                          /* Default, state-based abort. */
                          doAbort(*pendingList) ;
                          break ;
        }

        actionStatus = ABORTED ;
        currentAct = parentAction ;
      }

      return (actionStatus);
    }
```

Figure 6.6. Outline implementation for application-specific abort.

6.3. Supporting Object Replica Groups.

As well as considering the implementation of mechanisms used within the object model itself, some thought must also be given to the underlying communications support that will be required if objects are to be replicated. The type of communication mechanisms that are needed will depend upon the replication strategy that is to be employed. Further, the way in which those mechanisms are implemented will depend upon the properties and assumed failure modes of the communications medium. This makes a general discussion of the issues involved very difficult, so the approach adopted here will be to examine three specific implementations. First of all, the use of active replication to tolerate Byzantine faults in a system where processors are connected using point-to-point links. Secondly, the use of active replication to tolerate omission faults in a system connected over a broadcast network. Finally, the use of an alternative replication strategy, known as semi-active or leader-follower replication, to tolerate permanent omission (fail-silent) faults in a system connected over a broadcast network.

6.4. Active Replication : Fail-arbitrary Processing Nodes.

Consider a distributed system that consists of N processing nodes connected by means of a point-to-point communications network. If it is assumed that nodes, or the communications links connecting them, can exhibit Byzantine (arbitrary) failures, the only way to enhance the availability of objects within the system is to use some form of active replication. As described in chapter 3, this will entail the use of a suitable communications protocol, such as an atomic broadcast, to ensure that interactions between object replica groups are handled correctly. Such a protocol, developed by Cristian et al [Cristian 86], is described below.

6.4.1. Assumptions.

There are six basic assumptions upon which the implementation of the atomic broadcast protocol depends:

1. It is assumed that any processor or link failures that occur during the course of a broadcast leave the remaining processors in the system connected, in the sense that any two correct processors can still communicate over some route.

2. Network transmission delay is bounded. That is, if s and r are objects running on two correct processors, *s* and *r*, which are connected by a correct link, *l*, then a message sent from s to r over *l* is delivered to r and processed by r in at most δ time units. Note also that a distinction is made between reception of a message by a processor, *r*, and the delivery of that message to an object, r, running on that processor.

3. The clocks of correctly functioning processors measure the passage of time accurately and are synchronised, so that the measurable difference between the readings of correct clocks at any instant is bounded by a known constant, ε.

4. The operating system provides a "schedule(T,t,p)" command that allows a task T to be scheduled for execution at time t with parameters p. Multiple invocations of "schedule(T,t,p)" have the same effect as a single invocation and, if two tasks T_1 and T_2 are scheduled for times t_1 and t_2 ($t_1 < t_2$), then T_1 is started before T_2.

5. The clocks of correct processors are monotone (never set back) and no correct processor can issue the same timestamp twice, i.e. the granularity of time measurement is fine enough to discriminate between separate clock readings.

6. It is assumed that each processor, *p*, has a signature such that it is highly improbable that *p*'s signature could be forged by any other processor. Every processor also has access to an authentication procedure that allows it to verify the authenticity of a signature with a high probability. Methods for designing such signature and authentication schemes are discussed in [Rivest 78].

Given that these six assumptions are valid, it is possible to implement a *broadcast layer* that provides an atomic broadcast protocol capable of tolerating Byzantine faults in processors and communications links. The implementation can be broken down into three separate tasks that must be run on each processor in the system: a *Start* task that initiates broadcasts; a *Relay* task that ensures that broadcast messages are propagated throughout the system and an *End* task that delivers broadcast messages to their intended recipients. Note that throughout the following discussion, the term "request" refers to the contents of a broadcast, i.e. an application-level message (an invocation message, etc.) that the broadcast protocol is delivering, while the term "message" will be used to refer to the broadcast messages themselves, i.e. an application-level message plus its associated signatures, timestamps, etc. that are generated by the broadcast protocol.

6.4.2. The *Start* Task.

A high level description of this task is given in figure 6.7. An object that wishes to broadcast a request, σ, passes it to the local copy of the start task where it is received using the command "take(σ)". The broadcast of this request is identified by the local time, t, at which σ is received by the start task (line 5) and the identity, s, of the sender which is obtained by invoking the function "myid" (line 6). The identifiers returned by the function "myid" on distinct processors are guaranteed to be distinct and two broadcast identifiers, (t_1,s_1) and (t_2,s_2), are only equal if $t_1=t_2$ and $s_1=s_2$. Hence, by assumption 5, every correct processor generates broadcast identifiers that are unique system-wide.

```
1.   task Start
2.   const $\Delta = \pi(\delta + \varepsilon) + D_{\pi,\lambda} + \varepsilon$ ;
3.   var  t:Time ;
          $\sigma$:Application_request ;
          s:Sender_ID ;
          x: Signed_message ;
4.   cycle  take($\sigma$) ;
5.          t := clock ;
6.          s := myid ;
7.          sign(t,$\sigma$,x) ;
8.          send_all(x) ;
9.          H := H $\oplus$ (t,s,$\sigma$) ;        /* Update history */
10.         schedule(End,t+$\Delta$,t) ;
11.  endcycle ;
```

Figure 6.7. Atomic broadcast "Start" task.

After generation of the broadcast identifier, the message is signed by the processor using the "sign(t,σ,x)" primitive and transmitted on all outgoing links from the processor. The fact that a broadcast (t,s) of a request σ has been initiated is then recorded in a local history variable, H, that is shared by all of the broadcast layer tasks running on the node. Once this has been done, the *End* task is scheduled to start at local clock time t+Δ, at which time the value σ will be delivered to any destination objects resident on the node. Δ is the termination time for the protocol and it is given by:

$$\pi(\delta + \varepsilon) + D_{\pi,\lambda} + \varepsilon$$

where π and λ are the numbers of processor failures and link failures respectively, δ and ε are the network transmission delay and clock synchronisation accuracy defined earlier and the term, $D_{\pi,\lambda}$, is the *worst case message diffusion time* in the presence of at most π processor faults and λ link faults that do not disconnect the network. Strictly, $D_{\pi,\lambda}$ is equivalent to $\delta d_{\pi,\lambda}$, where δ is as defined above and $d_{\pi,\lambda}$ is the diameter of the surviving network, i.e. the greatest distance, measured along a shortest path, between any two nodes. However, note that for the purpose of calculating Δ, any upper bound on $D_{\pi,\lambda}$ could be used: for example $(n-\pi-1)\delta$.

6.4.3. The *Relay* Task.

The Relay task is shown in figure 6.8 and it works as follows. When a message is received, it is checked for authenticity (line 6). If the message has been corrupted, either by a faulty processor or a faulty link, it is discarded, Otherwise, the sequence of signatures appended to the message by other processors that have already accepted it is examined to ensure that there are no duplicates. If a duplicate signature is found, the message is discarded (line 7). (This is achieved using the **iterate** statement, which causes the task to abandon the remainder of its current execution cycle and to begin its next cycle - that is, to return to line 4 and begin processing the next message.) Also, since signatures are authenticated, the number of signatures appended to a message can be trusted and used as a hop count in determining its timeliness (lines 8,9 and 10).

```
1.   task Relay

2.   const $\Delta = \pi(\delta + \varepsilon) + D_{\pi,\lambda} + \varepsilon$ ;
3.   var  t,$\tau$:Time ;
          $\sigma$:Application_request ;
          s:Processor_ID ;
          x,y: Signed_message ;
          l:link ;
          S:Sequence_of_Processor_ID ;

4.   cycle  receive(x,l) ;
5.          $\tau$ := clock ;
6.          authenticate(x,t,$\sigma$,S) [forged : iterate] ;
7.          if   duplicates(S) → iterate ;        /* Duplicate.  */
8.          □    $\tau < t-\varepsilon|S|$ → iterate ;          /* Too early.  */
9.          □    $\tau > t+(\varepsilon+\delta)|S|$ → iterate ;     /* Too late.   */
```

```
10.          □   τ > t+Δ → iterate ;              /* Too late.   */
11.          fi ;
12.          s := first(S) ;

13.          if t∈dom(H) and s∈dom(H(t)) →
14.             if    H(t,s)  = σ → iterate ;    /* Already seen  */
15.             □     H(t,s)  = ⊥ → iterate ;    /* Faulty sender */
16.             fi
17.             H(t,s)  := ⊥ ;             /* "New" faulty sender  */
18.          □
19.             H := H ⊕ (t,s,σ) ;              /* New broadcast */
20.             schedule(End,t+Δ,t) ;
21.          fi ;

22.          co_sign(x,y) ;
23.          send_all_but(l,y) ;
24.  endcycle ;
```

Figure 6.8. Atomic broadcast "Relay" task.

Assuming that the incoming message is authentic, has no duplicate signatures and is timely, the history variable H is examined to determine whether the message is the first of a new broadcast. If this is the case, then H is updated with the information that the sender s (identified by the first signature appended to the message) has sent a new request σ at time t (line 19). The End task is then scheduled to start processing the received message at local clock time t+Δ (line 20) and the message is co-signed and forwarded on all links except the one upon which it arrived (lines 22 and 23).

If the message does not represent a new broadcast, there are three possible scenarios:

1. The received request may already be recorded in H because it has already arrived via an alternate route. In this case, the message should be discarded (line 14).

2. There may already be an entry in H for this particular broadcast, but specifying a different request. In this case, the sender must be faulty and this fact is recorded in H by setting the value associated with the broadcast to null, ⊥ (line 17). The message is then co-signed and forwarded (lines 22 and 23) so that other correct processors can also note the fault.

3. The final scenario is that the received request is associated with a broadcast that has already been identified as coming from a faulty sender. In this case, the broadcast will already be associated with the null value in H and the message can be discarded (line 15).

6.4.4. The *End* Task.

The *End* task (figure 6.9) is started on every correct processor at local clock time t+Δ to deliver the messages broadcast correctly at clock time t. If a unique request has been received for a broadcast initiated at time t then that request will be delivered to the appropriate recipients. If, on the other hand, two or more different requests have been received for a given broadcast, then the null value will be associated with that broadcast in H. This indicates that the sender must have been faulty and no request should be delivered. In either case, the details associated with the broadcast are deleted from the history, H.

```
task End(t:Time)

var p:Processor_ID;
    req:(Application_request ∪ {⊥}) ;

req := H(t) ;                /* Broadcasts made at time t.  */
while dom(req) ≠ ∅ do
  p := min(dom(req)) ;
  if req(p) ≠ ⊥ →            /* Only deliver if sender not  */
    deliver(req(p)) ;        /*   faulty, i.e. H(t,p) ≠ ⊥ . */
  fi ;
  req := req\p ;             /* Delete H(t,p) from H(t).    */
od ;
H := H\t ;                   /* Delete H(t) from H.         */
```

Figure 6.9. Atomic broadcast "End" task.

6.4.5. The Atomic Broadcast Protocol and Object Replica Groups.

The atomic broadcast mechanism described above provides all of the necessary communications support for active replica groups. Considering an object constructed according to the model proposed in chapter 4, there are two types of message that it may receive: *external messages* such as method invocations and *internal messages* such as the self-directed marker messages used in the implementation of the **select** construct (see figure 6.1). If all such messages, whether they are external or internal, are transmitted using the services provided by the broadcast layer, it can be ensured that all of the objects comprising an active replica group will receive the same set of messages in the same order. All replicas will, therefore, process the same set of requests in the same order and handle events such as method selection and the expiry of timing constraints consistently. State divergence will thus be avoided and tolerance of K Byzantine faults can be guaranteed using groups of 2K+1 replicas.

The replies generated by a replica group are handled as follows. For any given request, each replica generates its own reply and sends it, either using the atomic broadcast mechanism if the originator of the request was itself a replicated object or, possibly, using some form of reliable unicast if the originator of the request was a single object. The originator of the request can collate the replies that it receives and apply an appropriate majority voting algorithm to determine the correct response to its request. (Note that the same technique would also have to be applied to requests from a replicated client so that the server could determine the correct request to serve.)

6.5. Active Replication : Fail-silent Processing Nodes.

Where the above example considers a distributed system in which processing nodes communicate over point-to-point links and are subject to Byzantine faults, the example given here considers a distributed system in which processing nodes are connected by some form of broadcast network such as Ethernet or Token Ring. Further, it is assumed that processing nodes are fail-silent and that the network connecting them only exhibits bounded omission failures. The former assumption can be justified either by the use of appropriate fail-controlled processors or by using fail-controlled interface units to connect processors to the network.

The replication strategy to be examined will, once again, be active replication, so a suitable atomic broadcast protocol will be needed for communication with replicated objects. However, owing to the differences in the underlying system and its assumed failure modes, a different implementation of the atomic broadcast mechanism will be required. For example, there is no need for a node to relay broadcast messages to its neighbours because the initiator of a broadcast will be able to send network messages directly to each of the intended recipients. Also, message authentication is not necessary because the protocol only needs to be able to cope with omission faults.

6.5.1. Network Properties.

The protocol that will be used for this particular example is one version of the AMp atomic multicast protocol [Verissimo 89] that is used in the Delta-4 system [Powell 88]. This protocol is based on the two-phase commit principle described in [Gray 78] and it depends upon five basic network properties:

1. Although frames (i.e. network packets/messages) may be lost, when destinations receive a frame, they receive the one that was transmitted.

2. The number of consecutive transmission errors of a correct network is lower than some known value ω.

3. Any frame queued for transmission is sent on a non-faulty network within some known delay, T_{td}.

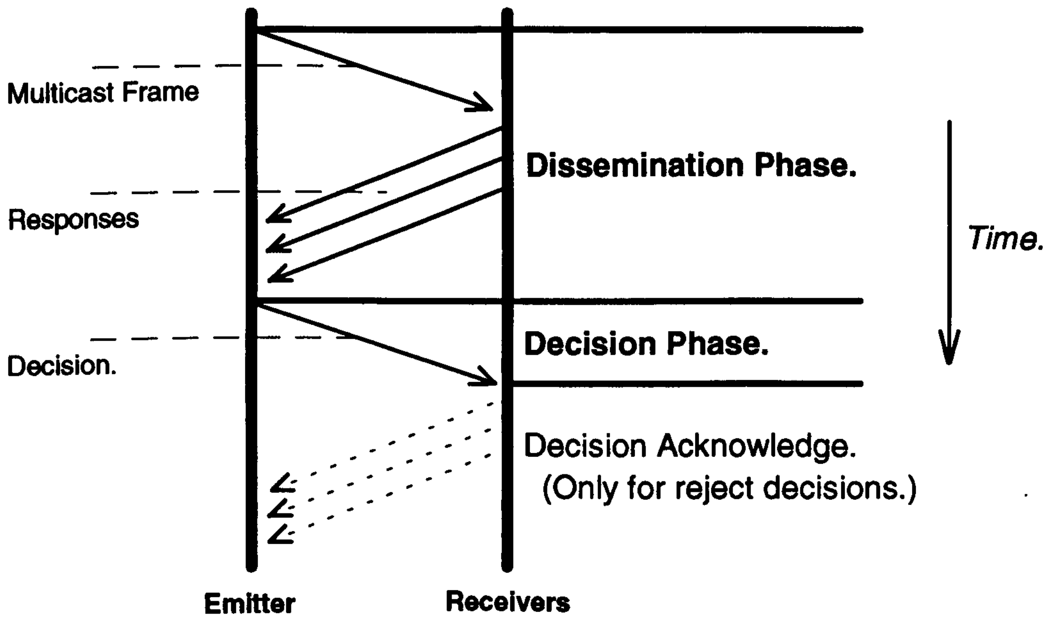
4. Any two frames received at any two sites are received in the same order. (This property is referred to as *Network Order.*)

5. The sender itself is also included in this ordering property as a recipient.

Although these five properties might, at first, seem difficult to achieve, they can be implemented in any ISO 8802 local area network or in the FDDI fibre-optic network and an implementation of the AMp protocol is already running on an 8802/4 Token Bus system.

6.5.2. Protocol Execution.

AMp works in terms of multicast groups, allowing the members of such a group to communicate with one another using the multicast protocol. To avoid confusion, throughout the following description the term "group" will be used to refer to such multicast groups, while the term "replica group" will be used when referring to the set of replicas of a replicated object.



Dissemination and Decision phases may be repeated up to ω +1 times.

Figure 6.10. The AMp Atomic Multicast protocol.

Given the network properties listed earlier, execution of the AMp protocol proceeds as follows. For each multicast group currently active on a node (i.e. each group that currently has a member executing on that node), several AMp communication entities are required. The AMp *Emitter Machine* (EM) and *Receiver Machine* (RM) are responsible for executing the AMp protocol for the multicast group and together constitute the *Group Communicator* (GC). The local *Group Monitor* (GM) executes certain critical functions related to the correct functioning of the multicast protocol, for example recording node failures so that a consistent local *Group View* (GV) can be maintained. Finally, the *Receive*

Queue (RQ) is managed by the receiver machine and holds any incoming messages for the multicast group.

To initiate a multicast to its group, the Emitter sends an information frame containing the broadcast message, implicitly querying the recipients if they can accept it. This *dissemination* phase is implemented using "transmission-with-response" rounds and ends after all of the expected responses have been received. In the worst case, this may take as many rounds as the omission degree of the network (ω) plus one and any node that does not respond within this number of rounds is assumed to have failed.

When the dissemination phase is complete, either because replies have been received from all of the intended recipients of the broadcast or because (ω+1) rounds have been completed, the emitter analyses the responses and issues its decision, either *accept* or *reject* (*decision* phase). The accept (proceed with broadcast) decision is only issued when unanimity can be reached. For example, if one of the intended recipients will not accept the message, the reject decision is issued and the message is rejected by all. Like the dissemination phase, the worst case duration of the decision phase is (ω+1) rounds. (The execution of this protocol is illustrated in figure 6.10.)

Failures during the decision phase are tolerated in two different ways, depending upon the decision that was made. If the decision was to reject the broadcast, then all recipients must acknowledge the rejection message and the lack of one or more acknowledgements will cause the emitter to re-transmit its decision (up to the re-try limit of the network - ω+1 attempts). As before, if a node does not reply to the decision message within the re-try limit, then it is assumed to have failed.

For an accept decision, the situation is quite different. When a group member receives the initial dissemination phase message, it sets an internal timer. If the timer expires before a decision has been received, an explicit decision request will be sent to the emitter of the broadcast. Since emitters, in the case of a reject decision, will only start a new transmission after assuring that all group members received the "reject" message, an emitter that receives a decision request can proceed as normal if it is still processing the broadcast in question or answer with an immediate accept if it is not. In the event of an emitter not responding to a decision request, the request will be re-transmitted until a reply is received or until the re-try limit is reached. In the latter case, it will be assumed that the emitter has failed and the group monitor will be called upon to terminate the broadcast and update the group view for the surviving group members.

6.5.3. Protocol Correctness : Assumptions and Order Properties.

The correctness of the AMp implementation described above depends upon three assumptions about protocol execution:

1. At any time, at most one AMp Emitter Machine is running at each node. (In fact, in the current implementation of AMp in the Delta-4 system, an emitter machine, once started, executes atomically.)

2. Each node runs as many AMp Receiver Machines as the currently active atomic multicast transmissions that it takes part in, each RM managing the state of the relevant Receive Queue.

3. If the decision for a broadcast is *Reject*, then the emitter positively confirms that it is received by all nodes belonging to its Group View. (Hence the use of positive acknowledgements and re-transmissions as described above.) This assumption allows an emitter to respond to decision requests as described above and avoids the need to maintain history lists for broadcasts.

Assumption 2 is a source of external parallelism in AMp, in the sense that the protocol will allow several concurrent executions for different groups to run simultaneously. Internal parallelism within a multicast group is ensured by allowing group members to run their Emitter Machines competitively in a fully decentralised fashion. Several transmissions from different nodes may be initiated simultaneously; order and agreement are then achieved by the network order property and the error detection and recovery mechanisms provided by the protocol.

Unlike the global ordering provided by the atomic broadcast protocol described in section 6.4, AMp provides incomplete orderings within multicast groups. All members of a group will observe the same set of messages in the same order and, if two messages m_1 and m_2 are causally related ($m_1 \rightarrow m_2$), then they will be delivered to the group in the appropriate order (m_1, m_2). However, if two messages are not causally related, they will be delivered in the same order to all members of the multicast group, but that order will not necessarily reflect the order in which the messages were originally transmitted. This is reminiscent of the causal ordering provided by the CBCAST primitive in the ISIS system [Birman 87][Joseph 88] and it is this property that allows parallel multicasts to different groups.

6.5.4. The AMp Protocol and Object Replica Groups.

The incomplete ordering property of AMp is sufficient for supporting active replica groups, since the fundamental criterion that all of the members of a replica group receive the same set of messages in the same order is still met. Communication with active replica groups could therefore be handled in the following way. An object that wished to transmit a request to an active replica group would join the existing multicast communication group that contained the replicas and multicast its request. Internal messages between replicas during the course of processing a request would be multicast to the group as a whole and would have to be accepted by the client object, but could subsequently be ignored. Then, when the request has been processed, each replica could either multicast its reply to the entire group or unicast its reply directly to the client. In either case, the client can proceed when it receives the first such response, since the system is assumed to exhibit only omission failures and any message received can, therefore, be assumed to be correct.

In the case of a replicated client, the procedure would be the same, but each client replica would join the multicast group and transmit its request independently. A server replica can simply respond to the first request that it receives and discard any duplicates that arrive later, while each client replica could ignore the duplicate service requests transmitted by the others. On completion of service, a multicast reply to the replicated client would be initiated by each replica in the server group.

6.6. An Alternative Strategy : Semi-Active Replication.

The last replication strategy that will be examined is the semi-active, or leader-follower, strategy used in the Delta-4 system and described earlier in chapter 3. Since semi-active replication requires fail-silent processing nodes, we assume a similar system to the one used in the previous example, however since semi-active replication only requires a reliable (atomic, unordered) broadcast communications mechanism, it will be assumed that the rel/$\text{REL}_{\text{r-atomic}}$ multicast protocol [Shrivastava 92a] is to be used for communication with the replica group.

6.6.1. The rel/$\text{REL}_{\text{r-atomic}}$ Protocol.

When a message, *m*, is multicast to a group *G* by a sender using this protocol, the following two receive-atomicity conditions are guaranteed:

1. If the sender completes the multicast, *m* is received by all functioning recipients in *G*.

2. If the sender crashes during the multicast, *m* is either received by all of the functioning recipients in *G* or it is received by none of them.

The protocol also ensures that successive multicasts from the same sender are received in the order in which they were sent, however this property is not important for the purpose of the example considered here.

The protocol consists of two layers. The lower layer, *rel*, provides a reliable multicast transport service for one-to-many communication. This service could be implemented in several different ways. For example, on a broadcast network such as an Ethernet, a multicast datagram service (unordered and unreliable), combined with acknowledgements and selective re-transmissions could form the basis for an implementation of *rel*. The higher layer of the protocol then makes use of this lower-level transport service to provide receive-atomic multicasts as follows. It is assumed that every processing node has a transmitter process (running in its network interface) to which objects wishing to perform multicasts

```
TRANSMITTER:
cycle
   get1(m)  → REL(m)
   ||
   get2(m)  → REL(m)
   ||
      ...
   ||
   getn(m)  → REL(m)
end cycle
```

Figure 6.11. rel/REL transmitter process.

can send their messages via FIFO (first-in, first-out) queues. The transmitter process has a number of concurrent threads which cyclically pick up messages from these queues (`get`$_i$`(m)`) and invoke a procedure, *REL*, for network transmission (see figure 6.11).

It is assumed that a message contains a list of destination addresses and a type field (type *first* for the first round of the protocol and type *second* for the second round). The algorithm for the procedure REL is then:

```
procedure REL(m:message)
{
     m.type := first ; rel(m)      /* First round.  */
     m.type := second ; rel(m)     /* Second round. */
}
```

Figure 6.12. REL procedure.

Every host also has a receiver process (figure 6.13) which is responsible for picking up messages. When the receiver process receives a new first round message, *m*, it creates a thread to monitor the progress of the multicast that gave rise to *m* and passes the message to this thread. The receiver process also accepts second round messages, which it passes to the appropriate threads, and *deathnotices*, which come from the threads that it has created.

Upon receiving a deathnotice from a thread that monitored the multicast for message, *m*, the receiver process will deliver *m* to the appropriate destination objects on its node and then act according to the value of the boolean, *successful*, returned by the thread. If the value "true" was returned, the multicast was completed successfully and the receiver process need take no further action. However, if the value "false" was returned, the multicast did not complete successfully and a new first round multicast for the message, *m*, will have been performed by the thread. The receiver process therefore has to complete the multicast by sending the corresponding second round message.

```
RECEIVER:
cycle
   receive(m)              /* Receive message from network. */
   case m.type of
      first:    if m is a duplicate → discard
                □
                m is not a duplicate → start a thread for m and
                                       deposit m in the queue
                                       for this thread.
                fi
      second:if m is a duplicate → discard
                □
                m is not a duplicate → deposit m in the queue
                                       of the thread for m.
                fi
   end case
||
   deathnotice(i) →        /* Thread i looking after m */
                           /*   has died, so... */
         deposit m in queues of m.dest objects on this host.
         if successful(i) → skip   /* Successful multicast.   */
         □
         not successful(i) → rel(m.type=second)
                                   /* Unsuccessful multicast */
                                   /*   so complete it.       */

         fi
end cycle
```

Figure 6.13. rel/REL receiver process.

A thread (see figure 6.14) that is created by the receiver process to monitor a multicast accepts the first round message that is passed to it and starts a timer. Two sub-threads are then created: one waiting for the second round message to arrive and one waiting for the timer to expire. If the second round message arrives in time, *successful* is set to true and the entire monitor thread is immediately killed. On the other hand, if the timer expires before the second round message arrives, the monitor thread will still be killed, but, first, a new first round multicast will be performed for the thread's message, *m*, and *successful* will be set to false. This behaviour, combined with the fact that a message will only be delivered to its local destination objects after the corresponding monitor thread has died, ensures that the

```
THREAD:
{
   get(m)              /* Get message from queue.       */
   start_timer(td)     /* Now, wait for second message. */
   do
      get(m) → return(successful=TRUE); die
                   /* Multicast successful.  */
   ||
      timeout → rel(m.type=first); return(successful=FALSE); die
                   /* Multicast unsuccessful. Start 1st round.  */
   od
}
```

Figure 6.14. Receiver process monitor thread.

message is guaranteed to be received at all other functioning destination nodes. Hence, if a sender crashes during a multicast, its message will either be delivered to all functioning destinations or it will reach none of them.

6.6.2. The rel/REL$_{\text{r-atomic}}$ Protocol and Object Replica Groups.

Use of the rel/REL$_{\text{r-atomic}}$ protocol with leader-follower replica groups is straightforward. Any object that wishes to send a request to a replica group passes the message to the local transmitter process running on its node, specifying the members of the replica group as the destination for the message. The multicast protocol then ensures that each member of the replica group receives the request (assuming that the sender does not fail during the multicast). Synchronisation messages sent by the leader of a replica group must also be transmitted using the rel/REL protocol to ensure that they are received by all of the followers. For objects such as those proposed in this thesis, identifying points at which synchronisation messages must be generated is easy. Possible points of non-determinism in such objects have already been identified in order to allow agreement and order protocols to be executed for active replica groups. Hence, all that is required for a leader-follower replica group is the replacement of the agreement protocols that would be used in such places with the appropriate synchronisation message protocol.

The treatment of replies in this example is also very straightforward: the replica group will only generate a single reply (from the leader) and this will be unicast back to the client (assuming that the client is not itself replicated). As with the previous example, if the client object was itself replicated, each client replica could multicast its request to the server group independently and the server replicas could discard duplicates. The reply would then be multicast to the client replica group by the leader of the server group.

6.7. Chapter Summary.

This chapter has considered the implementation of some of the mechanisms proposed for the object model that was described in chapter 4. A brief overview of the general structure of the model was given, showing that much of the basic functionality involved can already be found in existing programming languages and systems. As well as basic programming language constructs, this also included the exception handling mechanisms, timing constraint primitives and the method selection mechanism

A more detailed examination of the atomic action mechanism required by the model was then undertaken. In particular, the implementation of an atomic action construct that allows forward recovery operations was illustrated, based upon the existing atomic action construct found in the Arjuna system.

Finally, the chapter studied the communications support that is required for object replica groups. This consisted of an examination of three specific cases: the use of active replication to tolerate Byzantine failures in a point-to-point distributed system; the use of active replication in a system where fail-silent processors are connected by a broadcast network that is subject to omission failures; and the use of leader-follower replication in a similar broadcast-based system. In each case, a suitable communications protocol was described and the use of that protocol for communication with object replica groups was discussed.

Chapter 7.

Estimating the Cost of Replication.

Given that the object model proposed in chapter 4 can be implemented and that objects constructed according to that model can be replicated for availability, this chapter considers the overheads associated with the use of replication. In particular, it examines the issue of response times for replicated objects. In most cases, the response time for a replicated object will be slightly worse than that for a single, unreplicated object providing a similar service. This may be due to the need for sophisticated communication protocols, such as atomic broadcast, or it may be due to extra processing that a replica is required to perform, for example the construction and transmission of checkpoints (in the case of passive replica groups).

Since overheads will not only be different for different replication strategies, but also for different implementations of the same strategy, the approach taken here will be to consider three specific examples, each covering a different replication technique. In each case, theoretical worst case estimates will be derived for the response time of a replicated object, as compared with the corresponding single object. These estimates will be obtained by considering the extra functions and communications protocols that must be provided to support the replicated service. The resulting expressions will be in terms of system parameters, such as protocol latencies and scheduling delays, which must themselves be evaluated on a system-by-system or application-by-application basis.

The decision to consider worst case, rather than average case, estimates for the overheads of replication is due to the fact that we are primarily interested in real-time services. In a real-time system, it will often be necessary to predict worst case execution times for services, either to allow schedulability analysis to be performed, or to generate an actual schedule. Similarly, to permit guarantees to be given as to the continued provision of service in the presence of component failures, it will often be necessary to know the worst case response times for a replicated service.

Finally, another important issue that will be addressed in this chapter is the impact of replication upon the timing primitives used in the object model. In this case, the only replication strategy to have any major effect is active replication and processing overheads arise as a result of the need to ensure replica consistency on decisions regarding timing constraints. This is discussed in more detail below, before moving on to study the examples mentioned above.

7.1. Replication and Timing Constraints.

In passive and leader-follower replication schemes, the primary or leader can initiate recovery for timing errors as soon as they occur. The state of the application will then reflect the fact that the relevant timing constraint was not met and this view will be forced upon the other members of the replica group by means of checkpoints or synchronisation messages. In active replica groups, however, the situation is more complex. To maintain replica consistency, decisions regarding timing constraints must be agreed across the replica group and this requires communication between replicas. The temporal scope constructs that are used to express timing constraints must, therefore, be implemented in such a way as to allow the appropriate agreement protocol to be executed and this will have an impact on the time taken to initiate recovery for timing errors.

```
par
  send(self,"start") ; S ; send(self,"finish") ;
||
  select
    accept "start" → A ;
  □
    delay(EST) →
        select
          accept "start" →
                  select
                    accept "finish" → X ;
                  □
                    delay(EFT) →
                        select
                            accept "finish" → null ;
                        □
```

```
                                    delay(LFT) → Y ;
                                end select ;
                          end select ;
               □
                 delay(LST) → B ;
               end select ;
         end select ;
      end par ;
```

Figure 7.1. Deterministic implementation of temporal scope.

An implementation of the temporal scope mechanism that allows timing decisions to be agreed across a replica group has already been described in chapter three and is shown once again in figure 7.1. Two parallel threads are created, one sending a *start* message, executing the real-time operation S and then sending a *finish* message while the other thread awaits the arrival of the *start* and *finish* messages (with appropriate timeouts) using the **select...accept** construct. If the waiting thread uses a deterministic implementation of the **select** mechanism, timeout events can then be agreed and ordered across a replica group like any other type of message.

Assuming that the **select** construct is implemented using the generic input function as described in chapter three, the actual sequence of events when a timing constraint expires is as follows. The waiting thread sends a marker message to itself. If that marker message is received before the relevant start or finish message arrives, then the timing constraint is regarded as being missed. In an active replica group, marker messages and start and finish messages will be multicast throughout the group, validated (if necessary) and ordered like any other message. All replicas will therefore have a consistent view of whether or not a timing constraint has been met. However, in those cases where a timing constraint is agreed to have been missed, recovery for the timing error will not be able to begin until such time as the appropriate marker message has been received, validated and delivered. This introduces a delay, which in the worst case will be given by :

$$\Delta + T_V$$

where Δ = time for broadcast to group

and T_V = time taken to validate marker message.

The value of Δ will depend upon the multicast protocol that is used. For example, the AMp atomic broadcast protocol used in the Delta-4 system and described in the previous chapter can have a worst case execution time of:

$$\omega.[Tr(x) + Tr(1)] + T_{wd} + k.[Tr(x) + T_{gm}] + 2.\Gamma$$

where ω = number of network omission failures,

$Tr(x)$ = value of timeout set when sending a message and waiting for x responses,

x = size of multicast group-1,

T_{wd} = timeout at recipients waiting for decision from emitter (sender),

k = number of crash failures (of group monitors),

T_{gm} = timeout to detect failure of a group monitor and activate a new one,

and Γ = worst case network transmission delay.

The situation in which this arises is particularly pessimistic, assuming that :

i. The emitter only succeeds in transmitting its original broadcast message on its final attempt and then fails.

ii. The recipients wait to receive the decision message and must then query the emitter, deducing after ω attempts that the emitter has failed.

iii. There are k failures of active group monitors during the attempt to complete the broadcast and establish a new, consistent group view.

The value of T_v depends upon the assumed failure modes of replicas in the replica group. For Byzantine failures, the marker message can only be validated and delivered if a majority of the replicas send it correctly. In the absolute worst case, it is conceivable that a correct replica transmits its marker at some time, t, and that all of the other replicas in the group are slower than this replica by some bounded value, σ. The original transmitter will therefore have to wait until at least $t+\sigma$ for transmission of the other marker messages to commence. Assuming that the marker message only achieves a majority when the last member of the group transmits it, there may be a further delay of (N-1).Δ (where N is the number of group members and Δ is the time for a broadcast to the group) before the marker message can be properly validated and delivered. The overall worst case delay from the first correct replica detecting a timing error to the replica group being able to initiate recovery is, therefore :

$$\sigma + N.\Delta$$

(Note that Δ in this case would be for a Byzantine fault tolerant atomic broadcast, not for the AMp multicast protocol mentioned above.)

For absolute timing constraints (i.e. timing constraints specified using absolute rather than relative times), σ can be bounded by ε, where ε is the accuracy of clock synchronisation between processing nodes. For relative timing constraints however, this is not the case and σ will depend upon the time at which different replicas began execution of the relevant temporal scope operations.

For fail-silent active replicas, the value of T_V is implementation dependent. At one extreme, the first marker message may be taken to be valid and delivered immediately, in which case the first replica to detect a timing error would be able to initiate recovery after a delay of Δ. However, this also means that a timing error at one replica would cause the whole replica group to regard the timing constraint as missed. At the other extreme, it may be possible to implement a validation procedure for marker messages which ensures that the message is only delivered if a majority of the current group membership send markers. In this case, a single replica cannot dictate the outcome of a timing constraint to the rest of the replica group, but the worst case delay between first detection of a timing error at one replica and the initiation of recovery would then be :

$$\sigma + (\lfloor N/2 \rfloor + 1).\Delta$$

where σ = worst case de-synchronisation between replicas (as before),

N = current membership of replica group,

and Δ = time taken for a broadcast to the replica group.

Taking this particular example and assuming an absolute timing constraint, clock synchronisation between processing nodes of 8ms, five group members and a worst-case broadcast latency (in the presence of failures) of 15ms, this could represent a delay of the order of 53ms before recovery could be initiated for a timing error. Alternatively, taking the same figures for clock synchronisation and group size, but assuming a broadcast latency (with no failures) of 6ms, then the corresponding delay would be of the order of 26ms.

7.2. Response Times for Replica Groups.

As mentioned at the beginning of the chapter, response times for replicated objects may be worse than those for single objects that provide a similar service. In the following sections, this particular issue will be addressed in more detail by considering implementations of three specific replication techniques. First of all, the use of active replication in a system where processing nodes are fail-silent and connected by a broadcast network that suffers only omission failures. This is the same as the system that was considered in section 6.6 in the previous chapter and it will be assumed that the same broadcast protocol (AMp) is to be used. The second example is also taken from the previous chapter, being the use of leader-follower replication with the rel/$\text{REL}_{\text{r-atomic}}$ protocol as described in section 6.7. Finally, the third example will consider the use of passive replication in a similar LAN-based distributed system.

Before moving on to study these examples, it will be worthwhile to establish the baseline against which each one is to be compared. The worst case response time for a client to receive some service from a server in the singleton (non-replicated) case can be given by :

$$R_0 = 2.d + C_{max} + \theta$$

where R_0 = response time,

d = worst case message delay for a unicast,

C_{max} = worst case computation time at server (includes queuing time at server),

and θ = worst case descheduling of server by other tasks.

This is simply the time taken for the client to send its request to the server (d), the total time for the server to process the request ($C_{max}+\theta$) and the time taken for the server to send its response to the client (d). The extra term for descheduling (θ) represents the length of time during the processing of a request that the server is suspended while other tasks execute. This contribution to the overall service time has been included as an independent term because it is highly variable, being dependent upon the system scheduler and the other tasks or services that are running on the same processing node as the server. Corresponding terms will be found in all of the examples, where the impact of descheduling will be discussed at greater length.

Note that the above expression is for the response time in the absence of failures, since there is little to be gained by considering the response time for a failed server and a failure on the part of the network to deliver messages would simply entail re-transmissions, each of which would cause a further delay of d. In the following examples however, the failure of server replicas (up to the fault tolerance degree of the server) will be considered so that best and worst case response times can be estimated. In all cases, it will be assumed that the client is a single object and that the client does not itself fail. These assumptions ensure that the expressions obtained reflect the overheads due to replication at the server and allow a fair comparison to be made with the non-replicated case.

7.3. Active Replication on Fail-Silent Hosts.

We begin by examining one of the systems and replication strategies that was considered in the previous chapter. The system is based upon a broadcast local area network (such as Ethernet or Token Ring) that is assumed to exhibit only bounded omission failures and the host computers (processing nodes) connected to that network are assumed to be fail-silent. Given such a system, the continued availability of objects in the presence of K host failures can be ensured by having K+1 replicas running on different hosts. In this particular case, active replication will be assumed, with the AMp atomic multicast protocol being used to ensure the correct ordering of requests and messages at the replicas. In the absence of failures, the response time for the replicated service will be given by :

$$R_A(0) = \Delta + C_{max} + d + \theta_{resp}$$

where $R_A(0)$ = response time,

Δ= time to execute AMp atomic multicast protocol (transmitting request),

C_{max} = time for server to process request (including queuing time),

d = time for replica to unicast reply to client,

θ_{resp} = descheduling of server replica that responds to request.

Since it is assumed that the host computers in the system are fail-silent, the client can accept the first response that it receives and it is sufficient for each member of the replica group to unicast its reply directly to the client. Hence the service time for a request will consist of the time taken to broadcast the request to the server using the AMp protocol (Δ), plus the time taken to process the request by the first replica that responds ($C_{max} + \theta_{resp}$), plus the time taken for that replica to unicast its reply (d). Recalling that AMp is a two-phase protocol in which the first phase (*dissemination*) is performed using transmission-with-response rounds and the second phase (*decision*) is performed without acknowledgement (in the case of an *accept* decision), the termination time of the protocol in the absence of failures is given by :

$$\Delta = T_{resp}(N) + T_{dec}$$

where $T_{resp}(N)$ = time to transmit a network broadcast and receive N responses,

N = number of replicas in replica group,

T_{dec} = time to transmit decision message (time to send a network broadcast message).

Over a broadcast network such as an Ethernet or a Token Ring, it is likely that the time taken to transmit a broadcast or multicast network message will be comparable with the time taken to transmit a unicast. Hence, if we let d^* be the time for a broadcast over the network and if, during the dissemination phase, each member of the replica group unicasts its response to the client, the previous expression reduces to :

$$\Delta = d^* + N.d + d^*$$

$$= 2.d^* + N.d$$

$$\approx (N+2).d$$

Hence the response time for the replica group, in the absence of failures, will be :

$$R_A(0) = 2.d^* + N.d + C_{max} + d + \theta_{resp}$$

$$\approx (N+3).d + C_{max} + \theta_{resp}$$

7.3.1. Response Time with Failures.

For the system considered here, there are two types of failure that can occur: omission failures on the network and crash (fail-silent) failures of the host computers themselves. The former will increase the time taken to perform a multicast to a replica group and the time taken for replicas to send their replies to a client, while the latter will also be responsible for increasing the time taken to perform a multicast and may also have an impact, from the client's point of view, on the response times of individual replicas (if the failed replicas were the ones that had been giving the fastest responses).

If ω_1 and ω_2 are, respectively, the numbers of omission faults in the dissemination and decision phases of the AMp protocol, ω_3 the number of omission faults during the reply to the client and k the number of crash faults, the response time for the replica group becomes:

$$R_A(\omega_1,\omega_2,\omega_3,k) = \Delta(\omega_1,\omega_2,k) + C_{max} + (\omega_3+1).d + \theta_{resp}$$

where $R_A(\omega_1,\omega_2,\omega_3,k)$ = response time for replica group,

$\Delta(\omega_1,\omega_2,k)$ = AMp execution time with ω_1 dissemination omission errors, ω_2 decision omission errors and k failed recipients,

C_{max} = processing time at server replica,

$(\omega_3+1).d$ = time to unicast reply to client (ω_3+1 attempts, each taking d),

and θ_{resp} = descheduling delay at replica that provides first response.

In the event of omission errors, the AMp protocol proceeds by having the emitter time out and re-transmit the initial dissemination message until either it receives all of the expected responses or it has transmitted the message (Ω+1) times, where Ω is the omission degree of the network (i.e. maximum number of consecutive omissions). If, after (Ω+1) attempts, there are still responses missing, then those stations that have not responded are assumed to have failed. Since we are considering the situation in which one or more of the intended recipients has failed, the emitter cannot receive all of the expected responses and the initial message will be transmitted $(\Omega+1)$ times, regardless of the values of ω_1 and k.

For errors during the decision phase, each recipient sets a timeout when it receives the initial dissemination message. If a decision message has not been received before that timeout expires, the recipient will send an explicit "decision request" message, to which the emitter will reply with the appropriate decision. Since it is assumed, for the purposes of this example, that the emitter does not fail, a recipient that requests a decision will be guaranteed to receive a valid response after (ω_2+1) attempts, where ω_2 is the number of omission errors during the decision phase. In the worst case, this would entail $(\Omega+1)$ attempts like the dissemination phase. Hence, the overall time for the execution of AMp in the presence of failures is given by :

$$\begin{aligned}\Delta(\omega_1,\omega_2,k) &= (\Omega+1).T_{out}(N) + T_{wd} + (\omega_2-1).T_{out}(1) + T_{resp}(1) && \text{if } \omega_2 > 0\\ &= (\Omega+1).T_{out}(N) + T_{dec} && \text{if } \omega_2 = 0\end{aligned}$$

where $\Delta(\omega_1,\omega_2,k)$ = execution time of protocol

Ω = omission degree of network (maximum number of consecutive omission errors),

$T_{out}(N)$ = timeout set for transmitting a message and receiving N responses,

N = number of replicas in replica group,

T_{wd} = timeout at recipient (waiting for decision),

ω_2 = number of omission errors when requesting decision ($\leq \Omega$),

$T_{out}(1)$ = timeout set for transmitting a message and receiving one response,

$T_{resp}(1)$ = time taken to transmit a message and receive one response

and T_{dec} = time to transmit decision message.

If the time to transmit a broadcast message across the network is d^* and the corresponding time for a unicast message is d, the time for transmitting a message and receiving N responses will be $(d^* + N.d)$. Hence, the timeout set when transmitting a message and expecting to receive N responses must be at least this large and the previous expression, for the general case of $\omega_2 > 0$, can be reduced to :

$$\begin{aligned}\Delta(\omega_1,\omega_2,k) &= (\Omega+1).(d^* + N.d + \delta) + T_{wd} + (\omega_2-1).(2.d + \delta) + 2.d \\ &\approx (\Omega+1).(N+1).d + 2.\omega_2.d + T_{wd}\end{aligned}$$

if d^* is comparable to d and δ is small. This gives an overall service time for the replica group, in the presence of failures, of :

$$\begin{aligned}R_A(\omega_1,\omega_2,\omega_3,k) &= (\Omega+1).(d^* + N.d + \delta) + T_{wd} + (\omega_2-1).(2.d + \delta) + 2.d \\ &\quad + C_{max} + (\omega_3+1).d + \theta_{resp} \\ &\approx (\Omega+1).(N+1).d + 2.\omega_2.d + T_{wd} + C_{max} + (\omega_3+1).d + \theta_{resp}\end{aligned}$$

Further, T_{wd} must be at least $(\Omega+1).(d^* + N.d + \delta) + d^*$, since this is the time that it will take the emitter to transmit its decision, giving :

$$\begin{aligned}R_A(\omega_1,\omega_2,\omega_3,k) &= 2.(\Omega+1).(d^* + N.d + \delta) + d^* + (\omega_2-1).(2.d + \delta) + 2.d \\ &\quad + C_{max} + (\omega_3+1).d + \theta_{resp} \\ &\approx 2.(\Omega+1).(N+1).d + 2.\omega_2.d + d + C_{max} + (\omega_3+1).d + \theta_{resp}\end{aligned}$$

This expression is not dependent upon the number of replica failures (k) or the number of omission failures during dissemination (ω_1), because the dissemination phase of the AMp protocol will always require $(\Omega+1)$ transmission-with-response rounds if any of the intended recipients have failed. Similarly, the decision phase and the reply phase (i.e. transmission of the response to the client) only depend upon the number of network omission failures that occur during that particular phase. In the worst case, ω_2 and ω_3 will both be Ω, hence :

$$\begin{aligned}R_A(\Omega) &= 2.(\Omega+1).(d^* + N.d + \delta) + d^* + (\Omega-1).(2.d + \delta) + 2.d \\ &\quad + C_{max} + (\Omega+1).d + \theta_{resp} \\ &\approx 2.(\Omega+1).(N+1).d + 2.\Omega.d + d + C_{max} + (\Omega+1).d + \theta_{resp} \\ &\approx 2.(\Omega+1).(N+1).d + (3.\Omega+2).d + C_{max} + \theta_{resp}\end{aligned}$$

However, note that this situation will probably only arise in exceptional circumstances, since it assumes the maximum number of omission failures in all three phases of communication (dissemination, decision and reply). A more likely scenario for errors would be single failures in each phase or multiple failures in only a single phase, giving response times that are worse than the non-failure case, but better than the above.

7.3.2. Overheads and Reliable Networks.

Returning to the expression for the best-case response time, $R_A(0)$, derived earlier, it can be seen by comparison with corresponding expression for the non-replicated case, R_0, that the response-time overhead of communication with an active replica group, in the absence of failures, is given by :

$$\begin{aligned} O_A(0) &= 2.d^* + (N - 1).d \\ &\approx (N + 1).d \end{aligned}$$

where N is the number of replicas in the group, d^* is the time to transmit a broadcast message over the network and d is the time to transmit a unicast message over the network. On a network with a worst-case unicast transmission delay of, say, 3ms, this would represent an overhead of the order of 12ms for a group of 3 replicas or 18ms for a group of five. (Note that the possible difference in descheduling delay between the singleton object and the replicas of the replicated object has been ignored. This issue is discussed further in section 7.3.4.)

In the event of network omission failures, this response time degrades further. Given a network with an omission degree of Ω (i.e. at most Ω consecutive omissions), the worst case overhead increases to :

$$\begin{aligned} O_A(\Omega) &= 2.(\Omega + 1).(d^* + N.d) + d^* + (3.\Omega - 1).d + (3.\Omega - 1).\delta \\ &\approx 2.(\Omega + 1).(N + 1).d + 3.\Omega.d \qquad (d^* \approx d, \delta = 0) \end{aligned}$$

if there are omission failures during the decision phase of the broadcast to the replica group (i.e. $\omega_2 > 0$) and :

$$\begin{aligned} O_A(\Omega) &= (\Omega + 1).(d^* + N.d) + d^* + (\Omega - 1).d + (\Omega - 1).\delta \\ &\approx (\Omega + 1).(N + 1).d + \Omega.d \qquad (d^* \approx d, \delta = 0) \end{aligned}$$

if there are no failures during the decision phase (i.e. $\omega_2 = 0$). If Ω=2, d=3ms and there are omission failures during the decision phase of the broadcast protocol, this gives worst-case response time overheads for an active replica group of the order of 90ms for three replicas or 126ms for five replicas. Similarly, if Ω=2 and d=3ms, but there are no failures during the decision phase, the overheads would be of the order of 42ms for three replicas or 60ms for five replicas.

It is important to note that, since none of these overheads are dependent upon the number of replica failures but only upon the number of network errors, in cases where it can be assumed that the network is reliable and will always deliver messages correctly then the worst case overhead becomes the same as that in the non-failure case. This can be seen by setting Ω=0 in the previous expression :

$$\begin{aligned} O_A(\Omega = 0) &= (0 + 1).(d^* + N.d) + d^* + (0 - 1).d + (0 - 1).\delta \\ &= 2.d^* + (N - 1).d - \delta \\ &\approx (N + 1).d \qquad\qquad (\text{if } d^* \approx d, \delta = 0) \end{aligned}$$

This is, of course, one of the advantages of active replication in a system such as the one assumed in this example. So long as the group communication mechanism is reliable and does not lose or corrupt messages, response times for an active replica group will be more or less constant, regardless of how many replica failures have occurred. (Assuming that the fault tolerance degree of the replica group is not exceeded, i.e. that at least one functioning replica remains.)

7.3.3. Computation Times : C_{max}

An important point that has not yet been discussed in any detail is the difference in the value of C_{max} at different replicas. This term is used to represent the maximum execution time (including queuing time) for a given service request, *as measured while the server is running*. In other words, C_{max} is the time that it would take to process the request if the server was allowed to execute constantly without any interruption from other tasks. Using this definition, two objects that provide the same service should have similar values of C_{max}

for the same service request, so long as they have similar queues of requests to serve at the time. Now, since the members of an active replica group have identical message queues at all times, it follows that the value of C_{max} for a given request will be the same at each replica. Also, note that in the above calculation of overheads for the replica group, it has been assumed that the value of C_{max} for a request at a non-replicated object will be the same as that at the replicated object. While this assumption may not be entirely justified, it is likely that, for the same set of requests from the same set of clients, the values of C_{max} in the two cases would be comparable, since the arrival pattern and queuing of requests would be similar in both cases.

7.3.4. Descheduling : θ

Given that C_{max} only defines the time spent processing a request while the server is active, the importance of θ now becomes apparent. In most systems, it is possible that a server may be pre-empted or interrupted while processing a request. When this occurs, the server will be "descheduled". That is to say, its execution will be suspended for a finite time and some other task will be allowed to run. Broadly speaking, there are two possible sources of such descheduling: system service tasks which are executed by the underlying operating system to provide some specific function or to perform "housekeeping" activities and other application tasks (i.e. other clients or servers) running on the same processing node.

7.3.4.1. System Descheduling.

For an active replica group, it is likely that the amount of descheduling due to operating system activities will be approximately the same for each replica, since each processing node will be supporting similar operating system services. However, this degree of descheduling will not necessarily be the same as that for a single, non-replicated object. For example, consider the situation when communications protocols are executed on the host computers rather than on dedicated network interface units. If this is the case, processing nodes in a system that does not support replication will not necessarily need to run group communication protocols such as atomic broadcasts. Hence, descheduling due to the execution of such protocols will be observed in the replicated system, but not in the non-replicated system.

7.3.4.2. Application Descheduling.

The amount of descheduling due to other application tasks will not only be different for replicated and non-replicated objects, it will also vary across the replicas in a replica group. Given a fixed processing capacity (i.e. systems with identical sets of processing nodes), an application that uses active replication to enhance the availability of objects will load the system more heavily than the corresponding non-replicated application. The competition for processing resources at each node will therefore be greater and the degree to which application tasks are descheduled by other application tasks will probably increase.

Differences in descheduling across the members of a replica group will arise as a result of the differing load on each node in the system. At a given node, the probability of a server being descheduled during its execution will depend upon the number and priority of other tasks running on the same node and upon the scheduling algorithm that is being used. The length of time for which it is descheduled will then depend upon the scheduling algorithm and the execution characteristics of the other tasks (execution time, blocking, etc.). In the general case, although the scheduling algorithm that is used may be the same at all nodes, the distribution of tasks will be such that different nodes execute different task sets and replicas executing on different nodes will suffer different amounts of descheduling. This is the reason why a failure of some of the replicas in an active replica group can have an indirect effect on the response time of the group, as perceived by a client, since the failed replicas might have been those that were running on lightly loaded processing nodes and, therefore, producing the fastest responses.

7.3.4.3. Calculating θ.

Unfortunately, it is very difficult to calculate, or even estimate, values for the amount of descheduling experienced by a server. System descheduling depends upon the implementation of the underlying operating system and the set of services that it provides, while application descheduling depends upon the set of application tasks running on each node and the scheduling algorithm that is being used. Values for θ can, therefore, only be obtained on a per-system and per-application basis. For example, given a system such as the MARS system, where application tasks and their interactions across the network are all statically scheduled, it should be relatively easy to calculate the worst-case descheduling time for any task. On the other hand, given a highly dynamic system, such as CHAOS (which uses a dynamic thread scheduling algorithm), the worst case descheduling experienced by a given task could only be calculated by performing an extensive and complex analysis of the entire system.

Since descheduling is so dependent upon the detailed implementation of the system and the application task set, for the purposes of the analysis given here it has been assumed that it is approximately constant across a replica group and across replicated and non-replicated objects. While this is not necessarily true, there are circumstances in which it serves as a good approximation. For example, when considering the highest priority task in a particular system or when presented with a system in which all scheduling is static (as in MARS) and servers are allowed to execute requests to completion without being interrupted or pre-empted.

7.3.5. Timing Constraints.

In this particular example, it is also necessary to consider the impact of timing constraints on the response time of the replica group. As described earlier in section 7.1, if the members of an active replica group are to execute timing primitives in a consistent manner, the detection of timing errors must be made visible across the entire replica group using a message-based mechanism. This can introduce delays when timing errors occur because the replica group must agree that the timing constraint has been missed before recovery can proceed. Hence, although the response time of an active replica group may not be degraded further when it meets timing constraints, its response time will be increased when timing errors occur. The overheads that are involved are described in more detail in section 7.1, which also gives an expression for the associated multicast delay (in the presence failures) that can arise if using the AMp protocol that has been assumed in this example.

7.3.6. Partial Multicasts.

Before moving on to consider the next example, a final point must be made regarding failed and partial multicasts. The expressions that have been derived for response times in the presence of failures effectively consider the effect of a partial multicast to the replica group in which some of the recipients do not receive the message. However, there are two other possible types of failure that can occur with a multicast: a partial failure in which the sender fails and a failed multicast in which one of the recipients is not prepared to accept the message and the sender must transmit a reject decision to the rest of the group. Since the analysis presented here is mainly concerned with the response time for successful requests and it is assumed that the sender does not fail, neither of these cases have been considered. However, it should be noted that the occurrence of sender and multicast failures will have an indirect effect upon response times from the point of view of other clients and other requests. In the case of a sender failure, the activation of a group monitor to terminate the current multicast will affect the descheduling of other application tasks, while failed multicasts will make it necessary for a client to re-transmit requests and it will only be possible for the client to do so when the reject decision for the failed multicast has been transmitted. In the presence of failures, this can represent a delay of :

$$D_{fail} = (\omega_1 + 1).(d^* + N.d) + (\omega_2 + 1).(d^* + N.d)$$

where D_{fail} = time to terminate a failed multicast,

ω_1 = number of network omission failures before first "reject" response arrives (in worst case, $\omega_1 = \Omega$),

d^* = time to transmit a network broadcast message,

N = number of replicas,

d = time to transmit a network unicast and ω_2 = number of network omissions during transmission of "reject" decision (if a replica has failed, or in the worst case, $\omega_2 = \Omega$).

7.4. Leader-Follower Replication.

In this example, the system under consideration is the same as in the previous example, however a different replication technique will be assumed: *semi-active* or *leader-follower* replication. In this case, all replicas execute all requests, but only a single replica (the leader) sends a reply to the client and all non-deterministic events (input message selection, expiry of timing constraints etc.) are processed by the leader, whose decision is then forced upon the other members of the replica group by means of synchronisation messages.

Clients must multicast their requests to the replica group, however the only requirement for the multicast is that all recipients should receive the message and there is no need for ordering since this will be dictated by the leader. Hence, it will be assumed that the rel/REL$_{\text{r-atomic}}$ multicast protocol described in section 6.7 is to be used and the response time for the replica group, in the absence of failures, can then be given by :

$$R_{LF}(0) = L_r + C_{max} + d + O_{sync} + \theta$$

where $R_{LF}(0)$ = response time of replica group,

L_r = latency of rel/REL$_{\text{r-atomic}}$ multicast protocol,

C_{max} = processing time for service request at leader (includes queuing),

d = time to unicast reply to client,

O_{sync} = overhead for transmission of synchronisation messages

and θ = descheduling of leader during execution of service request.

This is simply the time taken for the request to be transmitted to the replica group (L_r), plus the total time for the leader to process the request ($C_{max} + O_{sync} + \theta$), plus the time for the leader to unicast its reply to the client (d).

The rel/REL$_{r\text{-atomic}}$ protocol consists of two rounds, each of which involves the sender transmitting its message using the *rel* reliable multicast service. In the absence of failures, the latency for the protocol, L_r, is therefore equal to :

$$L_r = 2.t_{rel}$$

where t_{rel} = execution time of *rel* reliable multicast.

On a reliable broadcast network that can be assumed not to lose or corrupt messages, t_{rel} would be equal to d^*, where d^* is the network transmission delay for a broadcast message. Alternatively, on a network that exhibits only bounded omission failures (as in the previous example), *rel* could be implemented using positive acknowledgements with re-transmissions. This would be similar to the transmission-with-response rounds used in the dissemination phase of the AMp protocol, giving an execution time of :

$$t_{rel} = d^* + N.d$$

in the absence of failures. The overall response time for the replica group, in the absence of failures, would therefore be :

$$\begin{aligned} R_{LF}(0) &= 2.d^* + C_{max} + d + O_{sync} + \theta \\ &\approx 3.d + C_{max} + O_{sync} + \theta \qquad \text{(if } d^* \approx d\text{)} \end{aligned}$$

for a reliable network and :

$$\begin{aligned} R_{LF}(0) &= 2.(d^* + N.d) + C_{max} + d + O_{sync} + \theta \\ &\approx (2.N + 3).d + C_{max} + O_{sync} + \theta \qquad \text{(if } d^* \approx d\text{)} \end{aligned}$$

for a network subject to bounded omission failures.

The term, O_{sync}, is intended to cover the overheads (at the leader) of sending synchronisation messages to the other members of the replica group. This particular aspect of the leader-follower replication strategy is discussed in more detail in section 7.4.5.

7.4.1. Response Times with Failures.

In the event of failures, the situation for leader-follower replication is more complex than that for active replication. If a follower fails, there is no impact on the service provided by the replica group, however if the leader fails, this must be detected by the followers and one of the followers must then assume the role of leader. This will introduce a slight delay in execution whenever a leader fails.

If it is assumed that the followers are ordered using some kind of ranking scheme (to allow a new leader to be elected quickly without having to run a complex election protocol), the response time for the replica group in the presence of leader failures becomes :

$$R_{LF}(k) = L_r + k.\phi + C_{max} + d + O_{sync} + \sum_{i=0}^{k} \theta_i$$

where $R_{LF}(k)$ = response time for replica group with k failures,

L_r = latency of rel/$REL_{r\text{-atomic}}$ protocol,

k = number of failures,

ϕ = worst-case failure detection latency (time for followers to detect failure of leader),

C_{max} = maximum computation time for request (includes queuing at first leader),

d = time to unicast reply to client,

O_{sync} = overhead for synchronisation messages

and θ_i = descheduling during execution at successive leaders (i = 0...k)

This corresponds to the response time in the absence of failures, plus an overhead for failure detection ($k \times \phi$ in the case of k leader failures), and with the descheduling at successive leaders taken into account. The total execution time for the request will still be C_{max}, since the followers execute any given request at the same time as the leader (see section 7.3.3), and the total overhead for synchronisation messages, O_{sync}, remains the same (see section 7.3.4).

The failure detection latency, ϕ, will depend upon the failure detection mechanism that is employed. In a system such as the one in this example, the easiest way to implement failure detection is by means of "I am alive" messages that are broadcast by the leader (or on the leader's behalf by its network interface) to the rest of the replica group at regular intervals. Since nodes are assumed to be fail-silent, the absence of such a message when one is expected can be taken as an indication of the failure of the leader. For a reliable network, the maximum failure detection latency using such a scheme would be equal to the time interval between successive messages, plus an allowance for clock synchronisation and message transmission across the network. The corresponding latency for a network subject to bounded omission failures would be equal to the time interval between successive messages, plus an allowance for clock synchronisation, plus an allowance for repeated message transmissions over the network (up to the network omission degree).

The latency of the rel/REL$_{r\text{-atomic}}$ protocol in the presence of failures depends upon the underlying network. For a reliable network, L_r would still be equal to $2.t_{rel} = 2.d^*$. This is a direct consequence of the assumption that the client does not fail during a request, since a partial broadcast will only arise with the rel/REL protocol if the sender fails and, in this case, the sender is the client. Hence, for a reliable network, the latency of the multicast protocol will be given by $2.d^*$, regardless of receiver failures.

For the bounded omission network and assuming the implementation of *rel* mentioned earlier, the latency of the protocol would be given by :

$$L_r = (\omega_1 + 1)(d^* + N.d) + (\omega_2 + 1)(d^* + N.d)$$

where ω_1 = number of omissions during first round

and ω_2 = number of omissions during second round.

(In the presence of any replica failures or for worst-case omission failures, both ω_1 and ω_2 would be equal to Ω, the omission degree of the network.)

7.4.2. Overheads.

Given the previous expressions for $R_{LF}(0)$ and $R_{LF}(k)$ and the corresponding expression for R_0, it can be seen that the response time overheads for communication with a leader-follower replica group, using the rel/REL$_{r\text{-atomic}}$ protocol on a reliable network, are :

$$O_{LF}(0) = 2.d^* - d + O_{sync}$$

in the absence of failures and :

$$O_{LF}(k) = 2.d^* - d + k.\phi + O_{sync}$$

in the presence of failures, where d^* is the time to send a broadcast message over the network, d is the time to send a unicast, k is the number of failures, ϕ is the latency of the failure detection mechanism that is used and O_{sync} is the overhead due to the transmission of synchronisation messages from the leader replica to the followers.

The corresponding response time overheads using the rel/REL$_{r\text{-atomic}}$ protocol over a bounded omission network are :

$$O_{LF}(0) = 2.(d^* + N.d) - d + O_{sync}$$

in the absence of failures and :

$$O_{LF}(k) = 2.(\Omega + 1).(d^* + N.d) - d + k.\phi + O_{sync}$$

in the presence of k replica failures, where Ω is the omission degree of the network and all other terms are as before.

In deriving these expressions for the overheads, note that two major assumptions have been made. First of all, that the value of C'_{max} for a leader-follower replica group is the same as that for a single, non-replicated object, even in the case where execution of a request is spread across several successive leaders. It is likely that this assumption may be reasonable in the absence of failures (i.e. when a single leader serves the entire request) and, as will be shown in the following section, it can also be justified in the presence of failures. The second major assumption is that the descheduling (θ) for a non-replicated server is the same as that for the leader-follower replica group, whether the request is served by a single leader or it is served by a succession of leaders, each of which is descheduled for an appropriate fraction of the overall time. This assumption is unlikely to be accurate, but it is impossible to calculate or estimate values for this parameter without considering a specific system and application.

Given the above assumptions and further assuming that $d^* \approx d$ and that O_{sync} is negligible (which may be the case in some systems), the following table gives order of magnitude estimates for the overheads of leader-follower replication (for d = 3ms, $\Omega = 2$ and ϕ = 15ms).

	3 replicas	**5 replicas**
Reliable : no failures	3ms	3ms
Reliable : worst-case	33ms	63ms
Omission : no failures	21ms	33ms
Omission : worst case	99ms	165ms

Figure 7.2. Approximate overheads for leader-follower replication.

7.4.3. Computation Times : C_{max}

Adopting the same definition for C_{max} as used in the previous example, the computation time for a given request at a leader replica should be comparable to that at a non-replicated object in most cases. However, in this example it has also been assumed that the computation time for a request will be the same if it is executed to completion by a single leader or if it is executed in consecutive sections by several successive leader replicas. For the purposes of the analysis shown here, this assumption can be justified, since :

i. Although the followers in a leader-follower replica group must lag slightly behind the leader in order for the synchronisation message mechanism to work, this delay is typically very slight.

and

ii. In the expressions for response times in the presence of failures, the worst case failure detection latency has been assumed for all failures. Hence, where a failure is detected quickly, the delay in execution at the new leader will be subsumed in the over-estimate for failure detection and, where a failure is only detected after the maximum failure detection period, the followers will have continued to execute while failure detection was performed and, therefore, have reached the same point in their execution as the failed leader. In either case, the total computation time for the request can still be taken as C_{max}.

7.4.4. Descheduling : θ

As in the previous example, it is impossible to calculate or estimate values for the descheduling experienced by an individual replica or by a non-replicated object without considering an entire system and a specific application. In the analysis presented above, it has therefore been assumed that the descheduling suffered by a leader replica during the course of a request will be similar to that suffered by the corresponding single object. It has also been assumed that, in the case of failures, the sum of the descheduling at a series of leaders is similar to the total descheduling that would be experienced by a single leader serving the same request to completion. Although these assumptions may be justified from the point of view of system descheduling, particularly where communications protocols are executed on dedicated network interface units rather than on the host computers themselves, both assumptions may prove to be inaccurate when considering descheduling due to other application tasks.

7.4.5. Synchronisation Messages : O_{sync}

Estimating the overhead due to the transmission of synchronisation messages by the leader in a leader-follower replica group is quite difficult. For input synchronisation messages, which are used to dictate the order in which requests should be processed, the situation is quite simple because there is one synchronisation message transmitted to all of the followers each time a new request begins execution. However, synchronisation messages may also be transmitted during the course of a request, either to inform the followers of the outcome of a non-deterministic choice made by the leader or to inform the followers that the leader has passed a particular *pre-emption* point (i.e. a point at which the execution of the current request might have been pre-empted to allow a higher priority request to be served). For this type of synchronisation message, it is much more difficult to give a general estimate of the overheads involved since the number of such messages is dependent upon the operation being performed.

Fortunately in some systems, such as the Delta-4 system, the overheads for transmitting synchronisation messages are negligible because the transmission is handled by a node's network interface unit on behalf of leader replicas running on that node. Alternatively, in cases where the leader must handle the transmission of such messages itself, but only input synchronisation messages are used, it becomes easier to estimate the overheads involved. In such cases, each request will generate one synchronisation message, which must be transmitted to all of the follower replicas. Assuming that the rel/$REL_{r\text{-atomic}}$ protocol is used to transmit the message, the overhead at the leader in the absence of failures will be $L_r = 2.t_{rel}$, while the overhead for completion of the multicast in the event of a leader failure during transmission will be $(t_{rel} + t_d) + 2.t_{rel}$, where t_d is the timeout set by a node while waiting for a second round message to arrive and t_{rel} is as defined earlier. (Note: t_d must be greater than t_{rel}.) These are the worst case overheads in the sense that they assume a simple implementation in which the leader handles the message transmission directly or creates a thread to do so (since the thread may simply deschedule the leader in order to execute). Also, note that an optimisation is possible when all but one of the replicas have failed, since the last remaining replica will not need to transmit any synchronisation messages at all.

7.4.6. Timing Constraints and Partial Multicasts.

In this example, the expiry of a timing constraint at the leader would cause a synchronisation message to be transmitted to the followers informing them of the timing error. However, once this message has been transmitted, the leader is free to initiate recovery. This is the only overhead associated with timing constraints for leader-follower replication and in cases where a timing constraint is met, no message need be generated at all.

For the purposes of this example, partial multicasts could not occur since it was assumed that the client did not fail and the client was the initiator of the broadcast. However, as in the previous example, it should be noted that when such partial multicasts do occur, they will have an impact on other clients in the system because it will be necessary for the remaining nodes to continue execution of the rel/REL protocol in order to complete the multicast. (As mentioned above, the latency of the rel/REL protocol when the sender fails is given by $t_{rel} + t_d + 2.t_{rel}$)

7.5. Passive Replication.

The last replication strategy to be considered here is passive replication, in which one member of the replica group (the primary) receives and responds to all requests, while the other members of the group act as passive backups to which the primary will periodically send checkpoints of its internal state. Then, if the backups detect the failure of the primary, a new primary is elected and resumes processing from its most recent checkpoint.

It will once again be assumed that the underlying system consists of fail-silent processing nodes connected to a broadcast network, however in this case it will be assumed that the network is reliable (i.e. does not lose or corrupt messages). It is also assumed that some kind of logical group addressing or name server mechanism is provided by the operating system to allow clients to obtain the address of the primary replica in a passive replica group. Clients can then unicast their requests to the appropriate primary, which will process the request and send its reply. At the end of each request, the primary will also checkpoint its state to its backups. Since it is important that all functioning backups receive each checkpoint, it will be assumed that the rel/REL$_{r\text{-atomic}}$ protocol used in the previous example to transmit requests and synchronisation messages is used in this example to transmit checkpoints. Further, since it is useful to make the operation of sending a checkpoint and transmitting a reply to a client atomic, it is assumed that both the checkpoint and the reply are sent in a single multicast, addressed to the client and all of the backups. The client can then simply discard the checkpoint information and the backup replicas can discard the reply (if appropriate).

In such a system, the response time for the passive replica group, in the absence of failures, will be given by :

$$R_P(0) = d + C_{max} + C_{cp} + L_r + \theta$$

where $R_P(0)$ = response time for replica group,

d = time to unicast request to primary,

C_{max} = processing time for request,

C_{cp} = time taken to construct a checkpoint,

L_r = latency of rel/REL$_{r\text{-atomic}}$ protocol (for transmitting checkpoint and reply),

and θ = descheduling of primary during execution of request.

Since it is assumed that the network is reliable, the latency of the rel/REL$_{r\text{-atomic}}$ protocol in the absence of failures will be :

$$\begin{aligned} L_r &= 2.t_{rel} = 2.d^* \\ &\approx 2.d && \text{(if } d^* \approx d\text{)} \end{aligned}$$

where t_{rel} = time for a multicast using the *rel* reliable multicast service,

and d^* = time to transmit a multicast message over the network.

giving an overall response time for the replica group of :

$$\begin{aligned} R_P(0) &= d + C_{max} + C_{cp} + 2.d^* + \theta \\ &\approx 3.d + C_{max} + C_{cp} + \theta && \text{(if } d^* \approx d\text{)} \end{aligned}$$

The time taken to construct a checkpoint (C_{cp}) depends upon the amount of state information in the server object and the checkpointing scheme that is used. This is discussed in more detail in section 7.5.5.

7.5.1. Response Times with Failures.

For the system considered in this example, a client would receive no response to its request if the primary replica failed while processing it. This situation arises because request messages are unicast to the primary and a new primary will, therefore, have no way of knowing which request the old primary was processing at the time of its failure. Under these circumstances, clients have to set a timeout while waiting for replies and simply re-transmit their requests if a response is not received. However, if it assumed that there is some mechanism whereby a primary can inform its backups of the next request that it will execute, for example by transmitting a copy of its outstanding request queue with each checkpoint, it may be possible for a new primary to begin processing the appropriate request immediately upon detecting the failure of its predecessor.

Even assuming the existence of such a mechanism, the response time for a passive replica group can degrade quite badly when failures occur. On the one hand, if the primary fails during the processing of a request, then the backup which becomes the new primary will have to commence execution from its most recent checkpoint, thus repeating any processing that had already been done, but not checkpointed, by the failed primary. On the other hand, if the primary fails during transmission of its reply and checkpoint, then the latency for the rel/$REL_{r\text{-atomic}}$ protocol will increase, since an attempt will then be made to complete the multicast.

As in the previous example, a failure detection mechanism will also be required to allow the backup replicas to detect the failure of the primary. Thus, considering the case where the primary fails before initiating a multicast for its checkpoint and reply, the response time for the replica group in the presence of k failures is given by :

$$R_P(k) = d + k.(\phi + C_{max} + C_{cp}) + C_{max} + C_{cp} + L_r + \sum_{i=0}^{k} \theta_i$$

with $L_r = 2.t_{rel} = 2.d^*$

where $R_P(k)$ = response time for replica group,

d = time to unicast request to primary,

k = number of failures,

ϕ = worst case latency of failure detection,

C_{max} = processing time for request,

C_{cp} = time to construct checkpoint,

L_r = latency of rel/REL protocol,

θ_i = descheduling at successive primaries ($i = 0...k$),

t_{rel} = time for *rel* multicast,

and d^* = time to transmit a broadcast message over the network.

This is very much a worst-case estimate for k failures, insofar as it assumes that each successive primary completes both the processing of the request and the construction of its checkpoint and then fails before initiating a multicast to transmit its checkpoint and reply message. Thus, the term $k.(\phi + C_{max} + C_{cp})$ represents the wasted execution time at each failed primary $(C_{max} + C_{cp})$, plus the worst case failure detection latency (ϕ), which is how long it may take for the backups to discover that the primary has failed in each case. To avoid the need to run an election protocol, it has been assumed that the backups have been ranked to determine the order in which they become primary (as with the followers in the previous example).

In the case where a primary fails during the transmission of its response and checkpoint (i.e. after initiating the multicast), the situation can become even more complex. The rel/REL$_{r\text{-atomic}}$ protocol attempts to terminate partial multicasts by completing them, rather than aborting them. This is achieved by having any recipients which receive a first round message, but not a second round message, effectively re-start the multicast. Hence, so long as a primary initiates a multicast, it is guaranteed that the multicast will complete, however if the node that is acting as a sender in an attempt to complete the multicast also fails, then a further delay will be introduced because the multicast will be started yet again elsewhere. The response time for the replica group in such situations therefore becomes :

$$R_P(k,f) = d + k.(\phi + C_{max} + C_{cp}) + L_r(f) + \sum_{i=0}^{k} \theta_i$$

with $L_r(f) = f.(t_{rel} + t_d) + 2.t_{rel}$
$\qquad\qquad = f.(d^* + t_d) + 2.d^*$

where $R_P(k,f)$ = response time with k primary failures and f multicast failures,

k = number of primary failures,

$L_r(f)$ = latency of the rel/REL protocol with f sender failures,

f = number of sender failures during multicast,

t_d = timeout set by receivers waiting for second round message ($t_d \geq t_{rel}$)

and all other terms are as before.

In this case, there is a saving of (C_{max}+C_{cp}) because the last primary to fail does not do so until it has initiated the multicast and the work that it has done is, therefore, not wasted. However, the latency of the multicast protocol is increased from $2.d^*$ to $f.(d^* + t_d) + 2.d^*$ where f is the number of sender failures during completion of the multicast. Note that f will be at least 1, since the primary that initiates the multicast is assumed to fail, and that (k + f) must be less than the fault tolerance degree of the replica group for the client still to receive a reply.

An important point to note in both of the above expressions is that the term for descheduling represents the descheduling experienced by each primary replica during the entire execution of a request. That is, the term $\sum_{i=0}^{k} \theta_i$ will be approximately equal to $(k+1).\theta$ where θ is the descheduling experienced by a single primary that processes the same request to completion. This is as opposed to the case for leader-follower replication, where each successive leader was assumed to experience a suitable fraction of the overall descheduling time, giving : $\sum_{i=0}^{k} \theta_i \approx \theta$.

7.5.2. Overheads.

Considering once again the response time for a non-replicated object, R_0, the response time overheads for a passive replica group, in the absence of failures, can be seen to be :

$$\begin{aligned} O_P(0) &= 2.d^* - d + C_{cp} \\ &\approx d + C_{cp} \qquad \text{(if } d^* \approx d) \end{aligned}$$

where it has been assumed that the descheduling experienced by the primary replica is similar to that experienced by a single, non-replicated object (see section 7.5.4). If d = 3ms and it takes 3ms to construct a checkpoint, this represents an overhead of the order of 6ms on each request.

In the presence of failures, the overheads involved will depend upon whether a primary fails while processing a request or while multicasting its response. In the former case, we have :

$$\begin{aligned} O_P(k) &= 2.d^* - d + C_{cp} + k.(\phi + C_{max} + C_{cp}) + \sum_{i=0}^{k} \theta_i - \theta \\ &\approx d + C_{cp} + k.(\phi + C_{max} + C_{cp}) + \sum_{i=0}^{k} \theta_i - \theta \end{aligned}$$

while, in the latter case, the corresponding expression is :

$$\begin{aligned} O_P(k,f) &= 2.d^* + f.(d^* + t_d) - d + \phi + C_{cp} + (k-1).(\phi + C_{max} + C_{cp}) + \sum_{i=0}^{k} \theta_i - \theta \\ &\approx (3.f+1).d + \phi + C_{cp} + (k-1).(\phi + C_{max} + C_{cp}) + \sum_{i=0}^{k} \theta_i - \theta \end{aligned}$$

where θ represents the descheduling experienced by a single, non-replicated server, all other terms are as defined earlier and it has been assumed in the approximation that $t_d \approx t_{rel}$.

If it is assumed that the descheduling experienced by each successive primary is the same as that for the single object (that is, $\theta_i = \theta$ for i = 0...k), then the above expressions reduce to :

$$\begin{aligned} O_P(k) &= 2.d^* - d + C_{cp} + k.(\phi + C_{max} + C_{cp}) + k.\theta \\ &\approx d + C_{cp} + k.(\phi + C_{max} + C_{cp}) + k.\theta \end{aligned}$$

and :

$$\begin{aligned} O_P(k,f) &= 2.d^* + f.(d^* + t_d) - d + \phi + C_{cp} + (k - 1).(\phi + C_{max} + C_{cp}) + k.\theta \\ &\approx (3.f + 1).d + \phi + C_{cp} + (k-1).(\phi + C_{max} + C_{cp}) + k.\theta \end{aligned}$$

Unfortunately, since these expressions include terms for descheduling, it is impossible to give estimated figures for these overheads without considering a specific application and system. However, if it is assumed that a primary is not descheduled while processing a request, which may be true in certain systems (e.g. in a MARS-like, static system), and

C_{max}= 250ms	k=1	k=2	k=3
f=0 : $O_P(k)$	274ms	542ms	810ms
f=1 : $O_P(k,1)$	30ms	298ms	566ms
f=2 : $O_P(k,2)$	39ms	307ms	575ms

Figure 7.3. Approximate overheads for passive replication (1).

appropriate values are assumed for the other terms, it is still possible to calculate order of magnitude figures. Figures 7.3 and 7.4 give such figures, for different values of k and f and assuming that d = 3ms, C_{cp} = 3ms, ϕ = 15ms and C_{max} = 250 or 500ms.

C_{max}= 500ms	k=1	k=2	k=3
f=0 : $O_P(k)$	524ms	1.042s	1.560s
f=1 : $O_P(k,1)$	30ms	548ms	1.066s
f=2 : $O_P(k,2)$	39ms	557ms	1.075s

Figure 7.4. Approximate overheads for passive replication (2).

Note the improvement in those cases where a primary fails after initiating its multicast rather than before. This is a result of the rel/REL protocol terminating multicasts by completion rather than abortion, since this ensures that the processing carried out by the failed primary does not have to be re-executed.

7.5.3. Computation Times : C_{max}

For the purpose of receiving and processing a request, the primary replica in a passive replica group can be regarded as being the same as a single, non-replicated object. Hence, for a given request, the value of C_{max} at a primary replica is likely to be the same as that at the single object. In the above analysis, it has also been assumed that, in case of failures, the value of C_{max} at successive primaries is the same. While this may not necessarily be true, it is likely to be an over-estimate of the required computation time rather than an underestimate because C_{max} at the first primary to serve a given request will probably include a greater contribution for queuing time than at subsequent primaries (which can commence execution of the current request immediately upon detecting their predecessor's failure).

7.5.4. Descheduling : θ

As in the other examples, this particular term is by far the most difficult to quantify without referring to a specific system and application. However, the assumption that the amount of descheduling suffered by a primary replica is approximately the same as that suffered by a non-replicated object is much more justifiable than the corresponding assumptions made for the active or leader-follower replication strategies. As far as application tasks are concerned, the load that a particular application places on a system will be approximately the same whether passive replication is used or not, since there is only one copy of any given object active at a given time. The difference in system descheduling is also likely to be slight, particularly if the multicast protocol for replies and checkpoints is executed by network interface units rather than the host computers.

On the other hand, the assumption that successive primary replicas will suffer a similar amount of descheduling during the processing of a given request is not so easily justified.

However, given a system in which processing load is balanced evenly across all hosts, it may still serve as a useful approximation.

7.5.5. Checkpointing : C_{cp}

The only term that remains to be considered is the overhead due to checkpointing, C_{cp}. Like descheduling, this term is also difficult to quantify, since it depends upon the type of checkpoints used and the times at which they are taken. Considering the type of a checkpoint, there are two basic options: either checkpoint the entire internal state of an object or only checkpoint those changes that have been made to the internal state since the last checkpoint was taken (*incremental checkpointing*). Typically, the former approach will generate larger checkpoint messages but the overhead for checkpointing will be more or less constant for a particular object. With the latter approach, smaller checkpoints will usually be generated, however the time taken to construct them will depend upon the operation that is being performed and the number of changes made to the internal state of the object.

The checkpointing strategy, in terms of the times at which checkpoints are taken, is also important. The approach used in this example, whereby a checkpoint is taken each time an operation is completed, is common in a number of systems and, typically, represents a reasonable trade-off between the overheads incurred and the amount of re-processing that must be performed by a new primary. If checkpoints are taken more frequently, a new primary will have less processing to do in order to bring itself up to date, but the overheads for checkpointing will increase (even though, for incremental checkpoints, the size of each individual checkpoint message will probably be smaller). Conversely, if checkpoints are taken less frequently, the overhead due to checkpointing will decrease, but a number of operations may have to be re-executed when a primary fails, thus lengthening the break in service which occurs at such times.

Finally, as with synchronisation messages in the leader-follower replication strategy, it is possible to optimise any checkpointing scheme when the fault tolerance degree of a passive replica group is reached, since there is then no need for the last surviving replica to transmit checkpoints.

7.5.6. Timing Constraints.

Timing constraints introduce no overheads at all in a passive replica group, since the primary can initiate recovery immediately upon detecting a timing error and the resulting application state will be reflected in the next checkpoint that is transmitted to the other members of the replica group.

7.6. Chapter Summary.

This chapter has concentrated on the overheads associated with the use of replicated services. First of all, the cost of executing timing primitives consistently across an active replica group was considered and it was shown that each replica will only be able to initiate recovery for timing errors after a short delay, during which the members of the replica group reach agreement as to whether the current timing constraint has, in fact, been missed.

The remainder of the chapter then studied response times for specific implementations of three different replication strategies: active, leader-follower and passive. In each case, an appropriate underlying system and suitable communications protocols were assumed, allowing theoretical expressions to be derived for the response time overheads associated with using a replicated, as opposed to non-replicated, service. The expressions obtained all included terms for application and system specific parameters such as descheduling delays and communications latency and, for each example, the factors affecting such parameters were discussed at some length. Each example also included estimated figures for the overheads involved, although these can only be taken as order of magnitude figures because, in each case, a number of simplifying assumptions had to be made to permit the relevant calculations to be made.

Chapter 8.

Conclusions and Further Research.

This chapter summarises the material that has been covered in this thesis, highlighting the contribution made by this work and suggesting further areas of research which have been identified.

8.1. Thesis Summary.

The thesis began with an introduction to the problems associated with providing fault tolerance in real-time programs. The use of objects and atomic actions as a structuring technique for real-time programs was briefly discussed and the basic concepts of exception handling and replication were outlined.

In chapter two, some of these areas were examined in more detail. The concept of a *real-time service* was introduced, including the notion of hard and soft services, and the issues that arise in the scheduling of such services were considered. The focus then switched to fault tolerance, with a more detailed discussion of the use of atomic actions as a structuring technique for fault tolerant programs. This was followed by a similar discussion on exception handling and the chapter closed with a short description and analysis of three existing fault tolerant, real-time systems.

Chapter three concentrated upon increasing the availability of services through the use of replication, with particular regard to its use in real-time systems. The various types of fault and failure that can occur in a distributed system were considered and the importance of maintaining consistency in replica groups was highlighted. Both passive and active replication strategies were then described, the latter in terms of the State Machine model. For both replication strategies, this included an examination of the types of fault that can be tolerated, the level of communications support that is required and the advantages and disadvantages of using that particular strategy. This led to a discussion of the relative suitability of the two strategies for use in a real-time environment, with the conclusion that active replication appears to be the better replication strategy for real-time use. The remainder of the chapter therefore addressed the problems associated with maintaining replica consistency in the presence of non-determinism. Possible sources of non-deterministic behaviour in real-time programs were identified and some existing approaches to dealing with such non-determinism were described. An alternative strategy for maintaining replica consistency was then proposed, based upon a set of application-level techniques that can be used to handle non-deterministic events.

The next part of the thesis continued the development of this idea, with chapter four describing a model for fault tolerant real-time objects. The general structure of the object model was outlined, before moving on to consider support for *active objects* and event handling. The inclusion of language-level support for timing constraints was discussed and an atomic action mechanism specifically geared towards the real-time applications domain was described. The provision of exception handling in the object model was also described and it was shown that, using the techniques described earlier in chapter three, objects constructed according to this model can be regarded as an extended form of State Machine for the purpose of replication. A brief comparison of the object model with two other existing models for real-time objects was then given and, in chapter five, the versatility of the proposed object model was illustrated by giving outline implementations of four different real-time applications.

In chapter six, some of the implementation and architectural support issues for the object model were examined. This included a short description of the way in which the general programming constructs used in the model could be implemented, as well as a more detailed description of the implementation of the atomic action and method selection mechanisms. The underlying support required for object replication was also considered and implementations of three specific replication schemes were described, demonstrating the level of communications support that would be needed and outlining the way in which object method invocations could be mapped to the appropriate communications primitives.

To conclude the thesis, chapter seven studied the overheads, in terms of response times, associated with the use of replicated objects. The cost of executing timing primitives consistently across an active replica group was examined and, finally, an analysis of the response times for replicated services under three different replication strategies (active, semi-active and passive) was presented.

8.2. Contributions of the Thesis.

It has been claimed that the next generation of real-time systems will need to be dynamic rather than static [Stankovic 88a]. That is to say, systems will increasingly need to adopt an event-driven approach to the real-time applications domain, as opposed to the time-driven approach that has been widely used to date in systems such as MARS. Considering the current growth in the use of computer systems for a wide range of different real-time control applications, it is likely that this claim will prove to be more than justified, however from the point of view of fault tolerance, a problem arises. One of the great strengths of the static, time-driven approach, alongside its predictability, is that it is relatively straightforward to make time-driven systems fault tolerant by using replication techniques. The use of replication to provide fault tolerance in event-driven systems, on the other hand, has not been extensively studied (with the exception of the work carried out in recent years within the ESPRIT Delta-4 project). This thesis has, therefore, concentrated on developing an object model that is well-suited for structuring event-driven systems and examined the impact of object replication on that model.

The major contributions of this thesis centre upon the development of the object model, in terms of providing support for replicated objects, and the analysis presented in chapter seven. In brief, the main points can be summarised as follows:

i) *Identification of the possible sources of non-determinism, and hence replica state divergence, in real-time programs.*

When replicating a service using an active replication strategy, care must be taken to ensure that uncontrolled non-determinism cannot arise and cause replica state divergence. However, real-time programs include several possible sources of non-deterministic behaviour, some of which are quite subtle and which may not be immediately apparent to the real-time applications programmer. This thesis has identified the five general classes of activity that can lead to such non-deterministic behaviour in a real-time application, thus allowing possible sources of replica state divergence to be recognised.

ii) *Proposal of mechanisms and techniques to deal with such non-determinism in real-time programs.*

This thesis has also proposed a small set of mechanisms and program structuring techniques that allows real-time programs to handle such non-deterministic activities in a way which still permits the use of active replication. The mechanisms and techniques proposed can be applied at the application level, or implemented as part of a programming language, and thus do not require specialised operating system support.

iii) *Development of a model for real-time objects.*

The mechanisms and techniques described above have been embodied in a model for fault tolerant real-time objects. This model includes a real-time atomic action mechanism that allows the definition of forward recovery operations. Objects can be subject to timing constraints, handle external events or system mode changes and contain internal threads of control, but they can still be regarded as State Machines for the purpose of replication and, hence, replicated for availability using active replication techniques.

iv) *Analysis of response time overheads for replicated objects.*

Finally, the thesis has explicitly identified the factors that contribute to the response time of a replica group, both in terms of system-level and application-level parameters. An analysis of the overheads for three different replication strategies has been presented, demonstrating the way in which estimated response times for a replicated service may be calculated in advance. This type of analysis could be used during the initial stages of the design of a system to calculate the degree of replication that can be supported for a given application or to scale the system in order to meet its fault tolerance requirements.

In conclusion, it cannot be emphasised too strongly that, unless special techniques like the thread scheduling mechanism adopted in the Delta-4 system are employed, the overheads discussed in the previous chapter are *unavoidable*. This can be regarded as the "negative side" of using replication to provide fault tolerance and it is important that the designers of current and future event-driven real-time systems understand this and take into consideration the constraints imposed by the use of replicated services.

8.3. Further Research.

The work described in this thesis only represents the first stages of the research that can be done into this topic and further development is possible in several areas. In particular, the following three aspects of this work are of special interest :

i) Further development of the object model.

There are some facets of the real-time object model that deserve further consideration. A particular case in point is concurrency control for replicated real-time actions. Although several effective concurrency control mechanisms have been developed for real-time transactions, none of them consider replicated transactions running on the members of a replica group. In such situations, different concurrency control decisions could be made at different replicas, leading to state divergence. At the moment, the object model prevents this happening by using only top-level actions and enforcing total mutual exclusion for concurrent operations. However, an interesting area of research would be to develop a concurrency control mechanism that allowed parallel actions to interact more freely, but which also guaranteed that the same concurrency control decisions would be made at different replicas.

ii) Implementation of the object model.

Perhaps the most important piece of work remaining to be done on the object model is an initial test implementation. This would involve the development of appropriate language support, either in the form of a suitable programming language and compiler or, more likely, in the form of a pre-processor and appropriate extensions to an existing language such as C++. It would also involve the development of the necessary run-time support systems, such as the atomic action and method selection mechanisms. (It is hoped that it will be possible to begin a partial test implementation sometime in the near future.)

iii) Further analysis of overheads.

In this area, there are two important tasks to be undertaken. First of all, simulation studies and, where possible, system tests should be carried out to test the validity of the expressions derived in chapter seven. Once again, it is hoped that the opportunity may arise to carry out such work sometime in the near future.

Secondly, a more complete analysis of the response time overheads for replica groups should also be attempted, using stochastic modelling techniques for example. In this way, estimates of average case, as well as worst case, response times could be obtained.

References.

[Ada 80]
U.S. Department of Defense, "Reference Manual for the Ada Programming Language", 1980.

[Agrawala 89]
A. K. Agrawala, O. Gudmundsson and D. Mosse, "Mission Critical Operating Systems Requirements and the MARUTI Project", University of Maryland Department of Computer Science Technical Report CS-TR 2342, November 1989.

[Anderson 76]
T. Anderson and R. Kerr, "Recovery Blocks in Action : A System Supporting High Availability", Proceedings of the 2nd International Conference on Software Engineering, 1976.

[Babaoglu 90]
O. Babaoglu, K. Marzullo and F. B. Schneider, "Priority Inversion and its Prevention in Real-Time Systems", PDCS Technical Report No. 17 (TR 90-1088), March 1990.

[Barrett 90]
P. A. Barrett, P. G. Bond et al, "The Delta-4 Extra Performance Architecture (XPA)", Proceedings of FTCS-20, Newcastle upon Tyne, 1990.

[Birman 85]
K. P. Birman et al, "Implementing Fault-Tolerant Distributed Objects", IEEE Transactions on Software Engineering, June 1985.

[Birman 87]
K. P. Birman and T. A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", 11th ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987.

[Birman 88]
K. P. Birman and T. A. Joseph, "Exploiting Replication", Cornell University Department of Computer Science Technical Report TR 88-917, June 1988.

[Birrell 84]
A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computers Systems, Vol. 2, No. 1, February 1984.

[Blazewicz 79]
J. Blazewicz, "Deadline Scheduling of Tasks with Ready Times and Resource Constraints", Information Processing Letters, Vol. 8, No. 2, February 1979.

[Bond 91]
P. Bond, D. Seaton, P. Verissimo and J. Waddington, "Real-Time Concepts", Chapter 5 in "Delta-4 : A Generic Architecture for Dependable Distributed Computing", Ed. D. Powell, ESPRIT Research Reports Series, Springer-Verlag, 1991.

[Burns 88]
A. Burns, "Scheduling Hard Real-Time Systems : A Review", University of Bradford Computer Science Report CS13-88.

[Campbell 86]
R. H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 8, August 1986.

[Chang 86]
H-Y. Chang and M. Livny, "Distributed Scheduling under Deadline Constraints : A Comparison of Sender-Initiated and Receiver-Initiated Approaches", Proceedings of 7th IEEE Real-Time Systems Symposium, 1986.

[Chen 78]
L. Chen and A. Avizienis, "N-version Programming: A Fault-Tolerance Approach to Reliability of Software Operation", Proceedings of FTCS-8, Toulouse, 1978.

[Cheng 86]
S-C. Cheng, J. A. Stankovic and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Real-Time Systems", Proceedings of 7th IEEE Real-Time Systems Symposium, 1986.

[Cheng 88]
S-C. Cheng, J. A. Stankovic and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems : A Brief Survey", IEEE Tutorial on Real-Time Systems, 1988.

[Chereque 92]
M. Chereque, D. Powell et al, "Active Replication in Delta-4", Proceedings of FTCS-22, Boston, 1992.

[Cristian 82]
F. Cristian, "Exception Handling and Software Fault Tolerance", IEEE Transactions on Computers, Vol. C-31, No. 6, June 1982.

[Cristian 86]
F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast : From Simple Message Diffusion to Byzantine Agreement", IBM Research Division Research Report RJ-5244, July 1986.

[Cristian 89]
F. Cristian, "Exception Handling", Chapter 4 in "Dependability of Resilient Computers", Ed. T. Anderson, Blackwell Scientific Publications, 1989.

[Damm 89]
A. Damm, J. Reisinger, W. Schwabl and H. Kopetz, "The Real-Time Operating System of MARS", ACM Operating Systems Review, Vol. 23, No. 3, July 1989.

[Dasgupta 91]
P. Dasgupta, R. J. LeBlanc, M. Ahamad and U. Ramachandran, "The Clouds Distributed Operating System", IEEE Computer, November 1991.

[Davidson 89]
S. B. Davidson, I. Lee and V. Wolfe, "Language Constructs for Timed Atomic Commitment", Proceedings of FTCS-19, 1989.

[Davidson 91]
S. B. Davidson, I. Lee and V. Wolfe, "Timed Atomic Commitment", IEEE Transactions on Computers, Vol. C-40, No. 5, May 1991.

[Dixon 88]
G. N. Dixon, "Object Management for Persistence and Recoverability", (PhD Thesis), University of Newcastle upon Tyne Computing Laboratory Technical Report No. 276, December 1988.

[Eswaran 76]
K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[Ezhilchelvan 90]
P. D. Ezhilchelvan, S. K. Shrivastava and N. A. Speirs, "An Examination of Fail-Stop Processor Architectures for Distributed Systems", University of Newcastle upon Tyne Computing Laboratory Technical Report.

[Frison 82]
S. G. Frison and J. H. Wensley, "Interactive Consistency and its Impact on the Design of TMR Systems", Proceedings of FTCS-12, 1982.

[Gheith 89]
A. Gheith and K. Schwan, "CHAOS^{art} : Support for Real-Time Atomic Transactions", Proceedings of FTCS-19, 1989.

[Gheith 90]

A. Gheith and K. Schwan, "CHAOS^{arc} : Kernel Support for Multi-Weight Objects, Invocations and Atomicity in Real-Time Applications", Georgia Tech. School of Information and Computer Science Technical Report GIT-ICS-90/06, January 1990.

[Goodenough 75]

J. Goodenough, "Exception handling, issues and a proposed notation", Communications of the ACM, Vol. 18, No. 12, December 1975.

[Gopinath 89]

P. S. Gopinath and K. Schwan, "CHAOS : Why One Cannot Have Only an Operating System for Real-Time Applications", ACM Operating Systems Review, Vol. 23, No. 3, July 1989.

[Gray 78]

J. N. Gray, "Notes on Database Operating Systems", in "Operating Systems : An Advanced Course", Lecture Notes in Computing Science, Vol. 60, Springer-Verlag, 1978.

[Halpern 84]

J. Y. Halpern, B. Simons, R. Strong and D. Dolev, "Fault Tolerant Clock Synchronisation", Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, August 1984.

[Haritsa 90]

J. R. Haritsa, M. J. Carey and M. Livny, "Dynamic Real-Time Concurrency Control", Proceedings of the 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, December 1990.

[Horning 74]

J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", Proceedings of the Conference on Operating Systems, IRIA, 1974.

[Ishikawa 90]

Y. Ishikawa, H. Tokuda and C. W. Mercer, "Object-Oriented Real-Time Language Design : Constructs for Timing Constraints", Proceedings of OOPSLA 90, Ottawwa, October 1990.

[Jensen 85]

E. D. Jensen, C. D. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", Proceedings of the 6th IEEE Real-Time Systems Symposium, 1985.

[Joseph 88]

T. A. Joseph and K. P. Birman, "Reliable Broadcast Protocols", Cornell University Department of Computer Science Technical Report TR 88-918, June 1988.

[Koenig 90]

A. Koenig and B. Stroustrup, "Exception Handling for C++ (revised)", Proceedings of USENIX C++ Conference, 1990.

[Kopetz 87]

H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems", IEEE Transactions on Computers, Vol. C-36, No. 8, August 1987.

[Kopetz 89]

H. Kopetz, A. Damm et al, "Distributed Fault-Tolerant Real-Time Systems : The MARS Approach", IEEE Micro, February 1989.

[Kopetz 90]

H. Kopetz and K. H. (Kane) Kim, "Temporal Uncertainties in Interactions among Real-Time Objects", Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, October 1990.

[Kramer 83]

J. Kramer, J. Magee, M. S. Sloman and A. M. Lister, "CONIC : an integrated approach to distributed computer control systems", IEE Proceedings (Part E), Vol. 130, No. 1, 1983.

[Kung 81]

H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981.

[Lamport 82]

L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", ACM TOPLAS, Vol. 4, No. 3, July 1982.

[Lamport 85]

L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", Journal of the ACM, Vol. 32, No. 1, January 1985.

[Lampson 81]

B. W. Lampson, "Atomic Transactions", Chapter 11 in "Distributed Systems - Architecture and Implementation", Springer-Verlag, 1981.

[Levi 89]

S-T. Levi, S. K. Tripathi, S. D. Carson and A. K. Agrawala, "The MARUTI Hard Real-Time Operating System", ACM Operating Systems Review, Vol. 23, No. 3, July 1989.

[Liskov 79]

B. H. Liskov and A Snyder, "Exception Handling in CLU", IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, November 1979.

[Little 90]

M. C. Little and S. K. Shrivastava, "Replicated K-Resilient Objects in Arjuna", Proceedings of IEEE Workshop on the Management of Replicated Data, Houston, Texas, November 1990.

[Little 92]

M. C. Little, "Object Replication in a Distributed System", (PhD Thesis), University of Newcastle upon Tyne Computing Laboratory Technical Report No. 376, February 1992.

[Liu 73]

C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, Vol. 20, No. 1, January 1973.

[Lomet 77]

D. B. Lomet, "Process Structuring, Synchronization and Recovery using Atomic Actions", ACM SIGPLAN Notices, Vol. 13, No. 3, March 1977.

[Melliar-Smith 82]

P. M. Melliar-Smith and R. L. Schwartz, "Formal Specification and Mechanical Verification of SIFT : A Fault-Tolerant Flight Control System", IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982.

[Mercer 90]

C. W. Mercer and H. Tokuda, "The ARTS Real-Time Object Model", Proceedings of the 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, December 1990.

[Mitchell 79]

J. Mitchell et al, "Mesa Language Manual", Xerox PARC Report CSL-79-3, 1979.

[Mok 83]

A. K. Mok, "Fundamental Design Problems for Distributed Systems for the Hard Real-Time Environment", Ph.D. Thesis, MIT/LCS/TR-299, 1983.

[Mosse 91a]

D. Mosse and A. K. Agrawala, "Resilient Computation Graphs for Distributed Real-Time Environments", University of Maryland Department of Computer Science Technical Report CS-TR-2613, February 1991.

[Mosse 91b]

D. Mosse, O. Gudmundsson and A. K. Agrawala, "The MARUTI System and its Implementation", University of Maryland Department of Computer Science Technical Report CS-TR-2694, June 1991.

[Muntz 70]

R. R. Muntz and E. G. Coffman (Jr), "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems", Journal of the ACM, Vol. 17, No. 2, April 1970.

[Nirkhe 90]

V. M. Nirkhe, S. K. Tripathi and A. K. Agrawala, "Language Support for the MARUTI Real-Time System", Proceedings of the 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, December 1990.

[Northcutt 87]

J. D. Northcutt, "Mechanisms for Reliable Distributed Real-Time Operating Systems : The Alpha Kernel", Perspectives of Computing Series Vol. 16, Academic Press, 1987.

[Parrington 88]

G. D. Parrington, "Management of Concurrency in a Reliable Object-Oriented Computing System", (PhD Thesis), University of Newcastle upon Tyne Computing Laboratory Technical Report No. 277, December 1988.

[PDCS 90]

Esprit Project No. 3092 (PDCS : Predictable Dependable Computing Systems) First Year Report, "Timeliness, Specification and Design for Dependability".

[Pease 80]

M. Pease, L. Lamport and R. Shostak, "Reaching Agreement in the Presence of Faults", Journal of the ACM, Vol. 27, No. 2, April 1980.

[Powell 88]

D. Powell, P. Verissimo et al, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", Proceedings of FTCS-18, 1988.

[Ramamritham 84]

K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems", IEEE Software, Vol. 1, No. 3, July 1984.

[Ramamritham 90]

K. Ramamritham, J. A. Stankovic and P-F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990.

[Rivest 78]

R. Rivest, A. Shamir and L. Adleman, "A Method of Obtaining Digital Signatures and Public-key Cryptosystems", Communications of the ACM, February 1978.

[Schlichting 83]

R. D. Schlichting and F. B. Schneider, "Fail-stop Processors : an Approach to Designing Fault-Tolerant Computing Systems", ACM Transactions on Computing Systems, Vol. 1, No. 3, 1983.

[Schneider 87]

F. B. Schneider, "The State Machine Approach", Cornell University Technical Report 86-800, December 1986, (Revised June 1987).

[Schneider 90]

F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach : A Tutorial", ACM Computing Surveys, December 1990.

[Schwan 87]

K. Schwan, P. Gopinath and T. Bo, "CHAOS : Kernel Support for Objects in the Real-Time Domain", IEEE Transactions on Computers, Vol. C-36, No. 8, August 1987.

[Schwan 90a]

K. Schwan, A. Gheith and H. Zhou, "From CHAOS^{base} to CHAOS^{arc} : A Family of Real-Time Kernels", Proceedings of the 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, December 1990.

[Schwan 90b]

K. Schwan and H. Zhou, "Optimum Preemptive Scheduling for Hard Real-Time Systems : Towards Real-Time Threads", Georgia Tech. College of Computing Technical Report GIT-ICS-90/28, September 1990.

[Sha 86]

L. Sha, J. P. Lehoczky and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling", Proceedings of the 7th IEEE Real-Time Systems Symposium, 1986.

[Sha 88]

L. Sha, R. Rajkumar and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases", ACM SIGMOD Record, Vol. 17, No. 1, March1988.

[Sha 91]

L. Sha, R. Rajkumar, S-H. Son and C-H. Chang, "A Real-Time Locking Protocol", IEEE Transactions on Computers, Vol. C-40, No. 7, July 1991.

[Shrivastava 82]
S. K. Shrivastava and F. Panzieri, "The Design of a Reliable RPC Mechanism", IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982.

[Shrivastava 90]
S. K. Shrivastava, P. D. Ezhilchelvan and M. C. Little, "Understanding Component Failures and Replication in Distributed Systems", ISA Project Report UNT/TR1, May 1990.

[Shrivastava 91a]
S. K. Shrivastava and A. Waterworth, "Using Objects and Actions to provide Fault Tolerance in Distributed, Real-Time Systems", Proceedings of 12th IEEE Real-Time Systems Symposium, San Antonio, Texas, December 1991.

[Shrivastava 91b]
S. K. Shrivastava, G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", IEEE Software, 1991.

[Shrivastava 92a]
S. K. Shrivastava and P. D. Ezhilchelvan, "rel/REL : A Family of Reliable Multicast Protocols for Distributed Systems", University of Newcastle upon Tyne Computing Laboratory Technical Report. (To be issued.)

[Shrivastava 92b]
S. K. Shrivastava and A. Tully, "Active Replication of Distributed Programs : Problems and Solutions", University of Newcastle upon Tyne Computing Laboratory Technical Report. (To be issued.)

[Sloman 87]
M. S. Sloman and J. Kramer, "Distributed Systems and Computer Networks", Prentice-Hall, 1987.

[Spector 83]
A. Z. Spector and P. M. Schwartz, "Transactions : A Construct for Reliable Distributed Computing", ACM Operating Systems Review, Vol. 17, No. 2, April 1983.

[Speirs 89]
N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing", Proceedings of FTCS-19, 1989.

[Srikanth 87]
T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization", Journal of the ACM, Vol. 34, No. 3, July 1987.

[Stankovic 88a]

J. A. Stankovic, "Misconceptions about Real-Time Computing : A Serious Problem for Next-Generation Systems", IEEE Computer, October 1988.

[Stankovic 88b]

J. A. Stankovic, "On Real-Time Transactions", ACM SIGMOD Record, Vol. 17, No. 1, March 1988.

[Stankovic 89]

J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", ACM Operating Systems Review, Vol. 23, No. 3, July 1989.

[Stroustrup 87]

B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1987.

[Tokuda 89]

H. Tokuda and C. W. Mercer, "ARTS : A Distributed Real-Time Kernel", ACM Operating Systems Review, Vol. 23, No. 3, July 1989.

[Tully 90]

A. Tully and S. K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, October 1990.

[Tully 91]

A. Tully, "Preventing State Divergence in Replicated Distributed Systems", (PhD Thesis), University of Newcastle upon Tyne Computing Laboratory Technical Report No. 328, June 1991.

[Verissimo 89]

P. Verissimo, L. Rodrigues and M. Baptista, "AMp : A Highly Parallel Atomic Multicast Protocol", Delta-4 Technical Report E89.076/D1/C, March 1989.

[Weinstock 80]

C. B. Weinstock, "SIFT : System Design and Implementation", Proceedings of FTCS-10, 1980.

[Wheater 90]

S. M. Wheater, "Constructing Reliable Distributed Applications using Actions and Objects", (PhD Thesis), University of Newcastle upon Tyne Computing Laboratory Technical Report No. 316, June 1990.

[Wolfe 90]

V. Wolfe, S. Davidson and I. Lee, "Supporting Real-Time Concurrency", IEEE Real-Time Systems Newsletter, Vol. 6, No. 2, Spring 1990.

[Wolfe 91]

V. Wolfe, "Supporting Real-Time Concurrency", Ph.D. Thesis, University of Pennsylvania, 1991.

[Zhao 87a]

W. Zhao, K. Ramamritham and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987.

[Zhao 87b]

W. Zhao, K. Ramamritham and J. A. Stankovic, "Preemptive Scheduling under Time and Resoure Constraints", IEEE Transactions on Computers, Vol. C-36, No. 8, August 1987.

[Zoubeir 91]

N. F. Zoubeir, "Fault-Tolerance Implementation for MARUTI, a Real-Time Distributed Operating System", University of Maryland Department of Computer Science Technical Report CS-TR-2728, July 1991.