The design of robust protocols for distributed real-time systems

Martin Robert Hill

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

September 1990

The University of Aston in Birmingham

The design of robust protocols for distributed real-time systems

Martin Robert Hill

Submitted for the degree of Doctor of Philosophy 1990.

Summary of thesis

Modern distributed control systems comprise of a set of processors which are interconnected using a suitable communication network. For use in real-time control environments, such systems must be deterministic and generate specified responses within critical timing constraints. Also, they should be sufficiently robust to survive predictable events such as communication or processor faults. This thesis considers the problem of coordinating and synchronizing a distributed real-time control system under normal and abnormal conditions.

Distributed control systems need to periodically coordinate the actions of several autonomous sites. Often the type of coordination required is the all or nothing property of an atomic action. Atomic commit protocols have been used to achieve this atomicity in distributed database systems which are not subject to deadlines. This thesis addresses the problem of applying time constraints to atomic commit protocols so that decisions can be made within a deadline. A modified protocol is proposed which is suitable for real-time applications.

The thesis also addresses the problem of ensuring that atomicity is provided even if processor or communication failures occur. Previous work has considered the design of atomic commit protocols for use in non time critical distributed database systems. However, in a distributed real-time control system a fault must not allow stringent timing constraints to be violated. This thesis proposes commit protocols using synchronous communications which can be made resilient to a single processor or communication failure and still satisfy deadlines.

Previous formal models used to design commit protocols have had adequate state coverability but have omitted timing properties. They also assumed that sites communicated asynchronously and omitted the communications from the model. Timed Petri nets are used in this thesis to specify and design the proposed protocols which are analysed for consistency and timeliness. Also the communication system is modelled within the Petri net specifications so that communication failures can be included in the analysis. Analysis of the Timed Petri net and the associated reachability tree is used to show the proposed protocols always terminate consistently and satisfy timing constraints.

Finally the applications of this work are described. Two different types of applications are considered, real-time databases and real-time control systems. It is shown that it may be advantageous to use synchronous communications in distributed database systems, especially if predictable response times are required. Emphasis is given to the application of the developed commit protocols to real-time control systems. Using the same analysis techniques as those used for the design of the protocols it can be shown that the overall system performs as expected both functionally and temporally.

Key words : Atomic commit protocols, Fault-tolerance, Petri nets, Real-time systems

Acknowledgements

I would like to take this opportunity to express my gratitude to everyone who helped me through my research. I am especially indebted to my supervisor Dr D.J.Holding for his guidance, persistence, and preparation with the thesis. I would also like to thank all my research colleagues in the department of Electrical and Electronic Engineering and Applied Physics at the University of Aston for their stimulating discussions and suggestions. A special thank you is due to my girlfriend, Becki, for being patient and helping me with the mundane tasks of day to day living as I prepared this thesis.

Finally I would like to thank the Science and Engineering Research Council and the University of Aston for funding throughout this research.

Table of contents

List of figures 7

Chapter 1 Introduction 9

1.1. Distributed computing systems 9

1.2. Coordination in distributed systems 12

1.3. Faults, reliability and safety 14

1.4. Summary of thesis 18

Chapter 2 Aims and Objectives of the research 21

2.1. Introduction 21

2.2. The coordination of distributed systems 22

2.3. Errors and faults 24

2.4. Software fault tolerance 26

2.4.1. Traditional software fault tolerance methods 27

2.4.2. Providing atomicity in the presence of faults 30

2.5. Design of robust commit protocols 31

2.5.1. Previous modelling of commit protocols 32

2.5.2. Petri net modelling of commit protocols 33

2.6. Implementation of commit protocols 34

2.7. Application of commit protocols to real time systems 35

2.7.1. Applications in real-time database systems 35

2.7.2. Applications in real-time control systems 36

2.8. Discussion 37

Chapter 3 Modelling atomic commit protocols 38

3.1. Introduction 38

3.2. Atomic commitment protocols 39

3.2.1. Blocking commit protocols 40

3.2.1.1. The two phase commit protocol 40

3.2.1.2. Variations on the 2 phase commit 41

3.2.2. Non-blocking commit protocols 44

3.2.2.1. The 3 phase commit protocol 44

3.2.2.2. Termination protocols 45

3.2.3. Independent recovery 46

3.2.3.1. Quorum based protocols 46

3.3. Finite State Machine models of commit protocols 47

3.3.1. The 2 phase commit FSM 47

3.3.2. The extended 2 phase commit 48

3.3.3. The 3 phase commit 49

3.4. Analysis of the FSM models 50

3.5. Problems with modelling commit protocols using FSM 54

Chapter 4 The design of robust atomic commit protocols 56

4.1. Introduction 56

4.2. Petri nets and time 58

4.2.1. Time Petri nets 60

4.3. Time Petri net models of commit protocols 61

4.3.1. Modelling the communication system 61

4.3.2. Two phase commit Petri net models 63

4.3.3. Extended 2 phase commit Petri net models 67

4.3.4. Possible Failures 70

4.3.5. Deadlock prevention 72

4.4. Failure and timing analysis of the models 73

4.4.1. Site failures 74

4.4.2. Link Failures 78

4.4.3. Timing analysis 84

4.5. Discussion 86

Chapter 5 Implementation and optimization of commit protocols
using synchronous communications 87

5.1. Introduction 87

5.2. Occam as a language for distributed systems 88

5.2.1. Occam 1 88
5.2.2. Occam 2 90
5.2.3. Other languages for programming distributed systems 91
5.2.4. Petri net models of Occam constructs 93
5.3. Occam implementation of commit protocols 94
5.4. Optimizations with synchronous communications 98
5.5. Discussion 105
Chapter 6 Application of commit protocols to real-time systems 107
6.1. Introduction 107
6.2. Characteristics of real-time systems 107
6.3. Commit protocols in real-time database systems 112
6.3.1. Example 1 : site failures 114
6.3.2. Example 2 : link failures 115
6.3.3. Summary 115
6.4. Applications in real-time control systems 116
6.4.1. Level crossing example 116
6.4.2. Drum and Slider example 126
6.5. Discussion 134
Chapter 7 Conclusions and further work 136
7.1. Conclusions 136
7.2. Further work 139
References 141
Appendix A 152
Appendix B 165

List of Figures

Fig.(3.1) FSM of 2 phase commit protocol 48

Fig.(3.2) FSM of extended 2 phase commit protocol 49

Fig.(3.3) FSM of 3 phase commit protocol 49

Fig.(3.4) Reachability tree for 2 phase commit protocol 50

Fig.(3.5) Reachability tree for extended 2 phase commit protocol 51

Fig.(3.6) FSM for extended 2 phase commit protocol resilient to site failures 54

Fig.(4.1) Example of a marked Petri net 58

Fig.(4.2) Reachability tree for fig.(4.1) 59

Fig.(4.3) Modelling communications 62

Fig.(4.4) 2 phase commit - asynchronous communications 64

Fig.(4.5) 2 phase commit - synchronous communications....................65

Fig.(4.5c) Reduced reachability tree for fig.(4.5a)....................66

Fig.(4.6) Extended 2 phase commit - asynchronous communications....................68

Fig.(4.7) Extended 2 phase commit - synchronous communications....................69

Fig.(4.8) Example site failure 70

Fig.(4.9) Example of a link failure 72

Fig.(4.10) Communication timeouts.................... 72

Fig.(4.11) Asynchronous extended 2 phase commit with site failures 74

Fig.(4.12) reachability tree for fig.(4.11) 75

Fig.(4.13) Synchronous extended 2 phase commit with site failures 76

Fig.(4.14) Reachability tree for fig.(4.13).................... 77

Fig.(4.15) Asynchronous extended 2 phase commit with link failures....................79

Fig.(4.16) Reachability tree for fig.(4.15) 80

Fig.(4.17) Synchronous extended 2 phase commit with link failures 82

Fig.(4.18) Reachability tree for fig.(4.17) 83

Fig.(4.19) Timing diagram of message transmission 85

Fig.(5.1) Two communicating Occam processes 89

Fig.(5.2) Petri net representations of Occam constructs 94

Fig.(5.3) Optiimized extended 2 phase commit 99

Fig.(5.4) Optimal extended 2 phase commit 102

Fig.(5.5) Petri net of 3 site protocol 104

Fig.(6.1) Example database system 115

Fig.(6.2) Outline Petri net of level crossing example 117

Fig.(6.3) Level crossing decision 118

Fig.(6.4) Petri net of level crossing with link failures 120

Fig.(6.5) Reachability tree for fig.(6.4) 121

Fig.(6.6) Partial reachability tree with second train at S1 122

Fig.(6.7) Partial reachability tree with second train at S2 123

Fig.(6.8) Petri net of level crossing with site failures 125

Fig.(6.9) Outline of arbor drum control system 126

Fig.(6.10) Slider motion 128

Fig.(6.11) Outline Petri net for slider drum control 129

Fig.(6.12) Petri net for slider drum control 131

Fig.(6.13) Reachability tree for fig.(6.12) 132

Chapter 1
Introduction

1.1. Distributed computing systems

Applications of modern computer systems require vast amounts of computing power, examples are image processing and weather prediction systems. Military image processing systems require very fast processing to process images in real time whereas a weather prediction system needs to analyse vast amounts of data. This necessitates the need to increase the speed of processing. To reduce the processing time three techniques can be applied either increasing the speed of the hardware, increasing the efficiency of the software or increasing the number of processors used.

The first solution will eventually reach the physical limits of a specific topology where it is impossible for a single processor to be made faster without radical changes in technology or architecture. Processing time is also affected by the efficiency of the algorithm being executed. Performance can be improved by restructuring the code and this is often carried out by a high level language compiler but this will also reach a limit. The third solution is that of parallel processing, this allows a task to be split over a number of processors all running in parallel.

Although the field of parallel processing is vast [Hwang 84] and beyond the scope of this thesis, it is useful to consider the applicability of various computer architectures to this research. A simple classification of computer architectures by Flynn [Flynn 66] identified four different structures by their instruction and data multiplicity. Although this scheme does not differentiate between input and output data the classification will suffice to demonstrate the multitude of architectures. The four structures identified were SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data stream), MISD (multiple instruction stream, single data stream) and MIMD (multiple instruction stream, multiple data stream).

SISD machines exhibit no parallelism and are typical of the usual Von-Neumann architecture. A SIMD machine performs the same instruction on a set of data concurrently, an example being an array processor. MISD machines are generally accepted as not having any applications [Basu 87] but this depends on the interpretation of instruction and data streams. A pipeline consists of one data stream (although different at each stage) being passed through a number of processing elements and could be considered as MISD. The final type, MIMD is more general purpose and consists of different instructions executing concurrently on different data.

One problem with the Flynn classification is that the processors are assumed to be tightly coupled, ie. they all share common memory. If no shared memory exists between the processors they are said to be loosely coupled.

Tightly coupled systems have the disadvantage that because memory is shared the physical distribution of the processors is limited. The reliability of the system depends on the shared memory because if the memory fails then no processors can communicate. Also as the distribution is limited it is likely that there are other common mode failures, such as failure of a shared power supply which would affect the whole system.

Loosely coupled MIMD architectures are commonly known as distributed processing systems. They consist of a number of autonomous processors executing asynchronously and communicating via a communication network. Additional processors can be added to such networks at will, thus providing additional computation power easily. The sites are assumed (but not necessarily) to be physically distributed. Thus a site failure (eg. power failure) or communication failure may not render the whole system unoperational, such failures are examined in section (1.3). Besides faster computation and resilience to failures, distributed processing systems also have the advantage of being able to naturally model many applications which are suited to concurrent processing.

With increased performance and versatility computers are increasingly being used to control applications where incorrect behaviour or component failures may have serious consequences, such as loss of life, environmental damage or damage to the application. Examples of such computer disasters have been collected by Neumann [Neumann 85]. Such systems are known as safety critical and must be designed so that the consequences of failures are minimized [Leveson 84, Leveson 86]. When a site or communication failure occurs in a distributed processing system it is possible for the remaining operational system to detect this and prevent any unsafe events.

This thesis examines the design of distributed real-time systems which generally comprise MIMD processing architectures. Emphasis is placed on the design of distributed systems which react in a safe manner when failures occur.

The processors in a distributed processing system are connected by a communication network which provides the facility for any two processors to communicate. The processors are also known as sites or nodes and in general may provide, communication functions, application functions or both. This thesis assumes that non of the sites in a distributed processing system provide purely communication functions. The communication network may comprise of various types of physical transmission media such as radio or satellite but is usually cable based. A communication link is defined to be the transmission path between two sites and may be a physical link or logical link. A physical link is a physical connection (circuit) between two sites whilst a logical link is an

The operations performed are not usually time critical although a reasonable response is expected. Recently interest has been shown in providing distributed database operations within real-time [Singhal 88, Lin 88] but this is linked to their use in control systems to store sensor information. It is therefore debatable as to whether a real-time distributed database is different to a real-time control system with a distributed database storage system. In either case, techniques to provide the desired properties and real-time response are required for safe operation.

A distributed control system obtains and sends information to its external environment by sensors and actuators. The application of distributed control systems are usually such that incorrect control is dangerous, eg consider the control of a nuclear power station or an industrial plant. A control system processes its inputs and provides suitable outputs to the system and operators. Control is often required within pre-defined time limits, systems providing control within temporal constraints are known as real-time control systems.

This thesis is concerned with generic aspects of real-time systems. It draws on the techniques developed in distributed database systems and distributed control systems and addresses the problem of designing robust real-time distributed control systems.

1.2. Coordination in distributed systems

Problems that occur in distributed systems can be classified as either coordination problems or problems due to faults and failures. Both type of problems occur in distributed databases and distributed control systems. Different solutions exist because databases operate on objects whilst control systems perform actions. Incorrect updates on objects can be corrected and are often insignificant but incorrect actions performed by a control system are rarely inconsequential.

To ensure the data contained in a database is accurate, correct and valid integrity constraints are used. They are assertions which are explicitly defined over a set of database resources (generally data items), for example A is always equal to B. They are used to guard against invalid updates and if they all hold the database is said to be correct. Another term often used for an integrity constraint is a consistency constraint, if all consistency constraints are satisfied then the database is termed consistent. Throughout this thesis the term integrity constraint will be used to mean assertions over data. The term consistency is used to mean that two or more values are in some kind of agreement unless otherwise defined.

Since the state of a database will change it must be ensured that that every user operation transforms the database into a new state which also satisfies the integrity constraints. The problem is to achieve this even in the presence of concurrent operations and failures. This prompted the design of transactions by Eswaren [Eswaren 76] which are now standard practice in database systems.

A transaction is an elementary unit of database access and may involve many fundamental operations on data such as read or write. It also provides the properties of atomicity, consistency, isolation and durability [Haerder 83]. Atomicity is the property that either all of the transactions operations are performed or none of them are. When a transaction completes, its updates are either committed to the database or undone. They are committed if the updates will not violate any integrity constraints, however if the integrity constraints are not satisfied the updates are undone, i.e. the transaction is aborted. The consistency property means that after the transaction has completed then the state of the database is such that all integrity constraints are satisfied. This property is often called serializability (see section (2.2)) when concurrent transactions are considered because serializability is a method of ensuring all the integrity constraints are satisfied. The isolation property prevents other concurrent transactions from seeing any results until the transaction has completed. This ensures a transaction does not have any side effects and can be safely restarted if necessary. The last property of a transaction, durability ensures that once a transaction has committed then the updates are not lost even if subsequent failures occur. These properties are required to aid the control of concurrent transactions and also recovery after failures. A user operation may consist of zero or more transactions but quite often only one, in such cases a user operation is termed a transaction.

Coordination is a problem in distributed systems because each site executes autonomously and does not know what the other sites have done without communicating with them. Thus communication is a vital part of the coordination. In addition distributed database systems can execute a number of transactions concurrently, either at one site or at many sites. The results of each transaction and the overall outcome must maintain a consistent database. To achieve this the execution of concurrent operations must be such that updates by one user are prevented from interfering with other database accesses. This allows transactions to be developed irrespective of their use in a single or multi user system.

Such a technique is known as a concurrency control mechanism and ensures that executing a number of transactions concurrently produces the same results as executing them in some serial order. However, this does not mean that a concurrency control mechanism enforces serial execution of transactions. Concurrency control prevents concurrent transactions from interfering with each other and producing incorrect results. It also restricts the availability of the resources so the granularity of the units used for concurrency control is very important when considering system performance. Concurrency control is a well developed field both for centralized and distributed systems [Bernstein 87a]. An early survey by Bernstein and Goodman [Bernstein 81] presents 48 different methods for concurrency control in distributed databases but still more have been developed [Bernstein 87b, Halici 89]. Most of these are variations on one method which has become almost standard, that is

2-phase locking. Other concurrency control mechanisms can be categorized as either being timestamp methods [Bernstein 80] or optimistic methods [Bhargava 83, Kung 81].

The coordination problem also appears in distributed control systems because the ordering and execution of concurrent events must be controlled. One example is when one event must occur before another, this is known as condition synchronization. Another problem is that of mutual exclusion where an event must be prevented from occurring whilst another is happening, this is usually used if two processes access the same resource. The solutions to these problems should be general purpose and allow as much concurrency as possible.

Techniques to solve these problems stem from operating system solutions [Andrews 83] and assume shared memory. Semaphores [Dijkstra 68] are a general purpose mechanism to solve these problems. A semaphore is a non-negative integer variable, s, associated with the resource which has two operations on it, P and V. P delays until s>0 and then executes s:=s-1 whilst V executes s:=s+1. To solve the mutual exclusion problem s is initially 1 and the first process to use s executes a P operation. The other processes cannot execute their P operations until s>0 which is provided by the original process executing a V operation (at the end). This is sometimes known as a binary semaphore.

Monitors [Hoare 74] are a similar technique but have a much more structured approach. A monitor consists of a set of permanent variables used to store resource states and a collection of procedures to perform operations on the resources. When a procedure within a monitor is called it is guaranteed to have mutually exclusive access to the required data. To solve the condition synchronization problem Hoare [Hoare 74] uses a variable which is acted upon in a similar manner to a semaphore.

Synchronization problems and concurrency control are similar because they both control the ordering of events. Concurrency control is in fact a series of synchronization problems with the ordering defined by the events involved, these change for each new set of transactions. Synchronization on the other hand is defined for a known set of events.

In this thesis techniques for providing the atomicity property of database transactions are extended for use in distributed real-time control systems. Holding, Hill and Carpenter [Holding 88] have proposed the use of such database techniques in real-time systems. This thesis concentrates on a method of designing the protocols so that the properties applicable to real-time systems can be demonstrated.

1.3. Faults, reliability and safety

Another problem with distributed systems apart from coordination is to maintain a reliable and safe system even in the presence of failures. The reliability of a system is a measure (usually expressed as a probability) of how its behaviour conforms to its specification over a period of time. The specification should be complete, accurate and unambiguous so that acceptable and unacceptable system behaviour can be distinguished.

A system failure is also related to the reliability of a system and is defined to occur when its behaviour deviates from that specified [Randell 78]. A failure is therefore an aspect of the system behaviour but they are caused by internal events known as errors which manifest from faults. A fault can be either a component fault or a design fault. and can be classified depending on its persistence. Three types are distinguishable, permanent faults, transient faults and intermittent faults. A permanent fault can appear at any time and remain until it is repaired. If a fault appears for a short period and then disappears it is known as a transient fault, an example being a fault caused by radio frequency interference. The final class of faults is the intermittent fault which recurs from time to time.

The previous definition of reliability can be applied to software systems but must assume that all inputs and hardware are fault free. If a system can tolerate such faults and retain a certain degree of functionality then it is said to be robust.

Unlike reliability the safety of a system is concerned with the actions of a system when it does not conform to its specification. A safe system is defined to be the free from conditions that cause human injury, environmental damage or costly repairs [Leveson 86]. As mentioned previously computers are being used more and more to control safety critical applications. Failures of such systems can be categorized as safety failures or non-safety failures [Leveson 86]. A safety failure is assumed to lead to one of the afore mentioned unwanted conditions. When a safety failure occurs it is preferable to prevent the dangerous action rather than attempting to provide correct system behaviour. Thus by ensuring the safety of the system its reliability may be reduced. A non-safety failure can be handled by a recovery mechanism that ensures all the critical functions of the system are maintained. If a recovery from a safety failure can not be made this way then a fail safe or fail soft procedure must be used. A fail safe system is such that when a failure occurs instead of attempting to maintain functionality the amount of damage caused by the failure is minimized, i.e. safety is of paramount importance. A fail soft system can continue operation with degraded performance or reduced functionality until the fault is removed.

This thesis is concerned with designing fail safe systems in a distributed real-time environment where various failures can occur. Emphasis is placed on the design of robust transaction protocols which will operate correctly in the presence of failures. A particular feature of the work is the imposition of real-time constraints on such protocols.

The nature of faults is different in hardware and software systems. Most hardware faults are due to components which deteriorate with time and so faults occur as the system ages. In comparison, software does not deteriorate with age and software faults arise from errors introduced at the design stage. Two types of faults must be tolerated, predictable hardware faults and unpredictable faults such as software design errors. This is sometimes termed fault-intolerance and fault-tolerance [Avizienis 75]. It is assumed that these faults can occur at any time.

The protocols designed in this thesis are designed to tolerate a clearly defined set of predictable hardware faults. The faults tolerated are permanent and persist until they are repaired. This thesis investigates the behaviour of transaction protocols when site and communication failures occur. The only site failures considered throughout this thesis are total site failures, i.e. a site stops processing until repaired. Communication failures can take many forms such as out of order messages, late messages or lost messages. The type of communication failure also depends on the type of communications used. This thesis considers a communication failure to be a failure of the communication media that prevents a number of sites communicating, this is also known as a network partitioning.

Reliable hardware is achieved through module design and re-use, the design being refined over a number of applications. However, software is often application specific and each new application introduces new design errors. Thus software is a primary cause of faults in a computer system. Two complementary techniques exist for reducing the effects of software faults, that is fault prevention and fault tolerance. Fault prevention attempts to reduce the number of design errors during development, whilst fault tolerance techniques are used to mask faults that do occur.

In an ideal software engineering environment fault prevention should not be necessary, the specification should map directly into code. This is practically impossible because even if the necessary error free compilers and translators existed there will always be some human decisions involved often at the specification stage. Specifications should be defined accurately and precisely to prevent any misinterpretations. Software engineering techniques [Sommerville 85] can be applied to prevent many design errors but errors will still occur. Since most software errors are due to incorrect interpretations of the specification [Kopetz 83] more and more emphasis is being placed on the use of formal methods in software engineering.

Software can still contain design errors even after rigorous development. To mask such faults software fault tolerance techniques must be used [Anderson 81]. A real application using such techniques has been shown to yield a substantial improvement in reliability [Anderson 85]. The first requirement for any fault tolerant mechanism is to detect the errors created by a fault and assess how much damage has occurred, e.g. how far errors have propagated. After this, recovery techniques must be applied, these are classified into backward or forward recovery techniques [Randell 78]. Backward error recovery restores a system to a previous state known to be free from errors. Forward error recovery usually transforms the erroneous state into a new correct state. In general forward error recovery is used for predictable faults whilst backward error recovery is used for unpredictable design faults. The two methods are complementary [Leveson 83] and can be used together.

A harder problem to solve is to provide fault tolerance in real-time systems [Anderson 85, Hecht 76, Bloch 89]. This is because the external environment is continually changing so techniques that use backward error recovery may restore a correct state which is out of date. Distributed systems also pose problems because errors may be propagated by inter-processor communication, such systems require mechanisms using the idea of atomic actions [Randell 75]. An atomic action in this case is defined as "a group of processes with no interactions between the group and the rest of the system for the duration of the activity".

A database system executes transactions which are atomic actions, however a transaction is different in that it can either abort or commit. If a transaction aborts then the system is restored to its original state which is consistent. If a transaction commits then the actions are performed and the database enters a new consistent state. Therefore a transaction differs from an atomic action in that it provides the additional property that either all its actions are completed or none at all, even if failures occur. This action is provided in database systems by an atomic commitment protocol.

Mancini and Shrivastava [Mancini 89] recently suggested that object and process model approaches to fault tolerance are duals of each other. They showed that replicated data management techniques can be useful for managing replicated processes. This thesis considers another aspect of this duality by considering the application of database atomic commit protocol techniques to control systems.

This type of database atomicity is often required in a distributed real-time control system but within a deadline. An example being the coordination of 2 robot arms to lift a container from a conveyor belt. The lift must be performed by both robots or neither, if only one robot attempts the lift the container will tip. Being able to tolerate faults is also important in such systems. Applying atomic commit protocols to real-time control systems is similar to providing transactions trimmed of certain properties. Stankovic [Stankovic 88b] suggested the use of transactions trimmed to provide a minimum set of properties for use in real-time databases.

This thesis proposes a new design for commit protocols which will provide atomicity within a deadline and also tolerate site and communication failures. Concurrent work by Davidson et al [Davidson 89] has also proposed the extension of a commit protocol with deadlines but their environment does not allow faults to be tolerated. The outcome of their commit protocol has another state in its outcome, the exception state, which is entered when a fault occurs, this indicates that recovery must be performed.

This thesis develops a new method of designing robust commit protocols which are suitable for use in distributed real-time control systems. The design is performed by using a formal model which is developed to study the effects of site and communication failures. The model used also includes timing information so that timing properties of the commit protocols can be investigated. Since failures are allowed the commit protocols are extended with mechanisms to detect and tolerate them. The correctness of the commit protocols developed can be shown by analyzing the formal models.

1.4. Summary of thesis

Chapter 2 outlines how the objective of designing robust distributed real-time systems which are fail-safe was realized. Firstly the problems of distributed systems are classified as being caused by the effects of concurrency or failures. The different techniques used to solve these problems in the database domain and control field are surveyed with comparisons drawn where possible. The techniques used by control systems to tolerate faults are shown to be deficient of mechanisms that provide the atomicity property when failures occur. It is proposed that a database technique, namely commit protocols, can be extended and applied to distributed control systems to provide atomicity. To be of use in real-time systems commit protocols must be extended with a deadline and also shown to operate correctly. Therefore an aim of this thesis is to develop a method of designing commit protocols that can show timing and functional properties are satisfied. The model should also be easily transformed into an implementation.

A commit protocol must provide as its result a set of states which are consistent with each other. To show such consistency a model of commit protocols that represents state is required. A survey of previous work on the modelling of commit protocols is provided in chapter 3. By including failures in the specification it can be seen where mechanisms are required to increase the resilience of the protocols. This chapter also shows why the previous modelling techniques are inadequate to design commit protocols for use in real-time systems. In particular time is omitted and it is therefore impossible to verify timing properties using these models. The model also omits the communication system and so communication failures can not be modelled. Many different commit protocols exist, to examine their usefulness to real-time systems they are surveyed in chapter 3. It is shown that protocols can be classified into those that need to communicate after a failure and those that can recover independently. The latter class of protocols are shown to be of more use in real-time systems because timeliness can be preserved.

Since previous models are inadequate to show timeliness and tolerance to communication failures, a new design technique must be developed. This thesis describes in chapter 4 how Petri nets [Peterson 81] can be used to solve the problems of the previous techniques. It is shown how commit protocols can be modelled with the communications included.

This allows protocols to be designed and analysed using both asynchronous and synchronous communications. The protocols are analysed under fault free conditions and shown to be correct. Various failures are then included in the model and the protocols enhanced to tolerate such failures. Including the communication system in the model allows communication failures to be studied. The commit protocols using asynchronous communications are shown to be directly equivalent to the previous models. However the commit protocols using synchronous communications are different because they can be made resilient to a communication failure as well as a site failure. This is because the sender always knows if a message has been received or not, unlike asynchronous communications. The new commit protocols using synchronous communications cannot take advantage of previous techniques for timeout placement and so a new method is developed by analysing the Petri net with failures included. Timing analysis can be performed by including time in the Petri net model. This permits the timeout values to be estimated directly. This calculation of timeout values is provided in greater detail in section (6.4.2).

Chapter 5 outlines how the robust commit protocols developed in chapter 4 can be implemented using a distributed programming language. Occam is used to demonstrate the implementation of the commit protocols because it includes primitives for synchronous communication and parallel processing and is also uncomplicated. It is shown how additional assembler routines must be used when failures are expected because Occam has insufficient semantics. Chapter 5 also shows how the new protocols using synchronous communications can be optimized by reducing the number of explicit messages sent. This is achieved by allowing the synchronous acknowledgement to infer information. So far in this thesis, for simplification, the protocols have only been using two sites. It is shown in this chapter how the optimized protocol can be extended to more than two sites.

Applications of the new robust commit protocols developed are described in chapter 6. The first application is a real-time database, this shows how the commit protocol can be used to ensure consistency even if when communications fail. It is also proposed that allowing the commit protocol to abort a transaction within a deadline is better than waiting indefinitely before committing a transaction. This would allow other transactions which are possibly more time critical the opportunity to complete in time. The protocols are then applied to two control examples, both illustrating the need for the all/nothing property. The second control example also requires timing constraints to be met, in particular that the commit/abort decision is always made within a deadline. The control examples are modelled using Petri nets so that the commit protocol models can be directly incorporated. This allows the analysis of the system to follow in a similar manner to the analysis of the commit protocols. It is shown how consistency is maintained. In the second example timing constraints are added to the Petri net model which is then analysed to show that a consistent decision is reached within the required deadline. It is also shown in greater detail than in chapter 4 how bounds on the timeout values can be calculated directly from the Petri net model of the system.

Finally the achievements of the research are summarized in chapter 7. Conclusions are drawn about the use of commit protocols in control systems and the modelling technique used. A section suggesting further work is also provided.

Chapter 2
Aims and objectives of the research

2.1. Introduction

This chapter outlines the aims of the research and examines existing work and literature in the field of distributed systems. Two aspects of distributed systems are considered in this chapter, the control of concurrent operations and their resilience to faults. The scope of this chapter includes a study of coordination techniques in section (2.2), an examination of errors and faults in section (2.3) and a review of traditional fault tolerant techniques in section (2.4).

A distributed computing system comprises hardware and software and so faults can occur in either. Hardware faults are mainly due to deterioration with age whereas software faults are caused by design errors. It is desirable for the coordination of distributed systems to be able to tolerate such failures. If a failure cannot be tolerated then this may result in an undesirable state which could possibly be unsafe. The problem of coordinating distributed systems in the presence of faults is examined in section (2.4.2). This leads to the identification of a class of protocols which are ideal for such coordination. However, the protocols require enhancements if they are to operate correctly in distributed real-time control systems in the presence of faults.

Models exist which have been used to ensure the protocols functioned correctly. Existing models are examined in section (2.5.1) and their advantages and limitations are discussed. This analysis leads to the conclusion that these models cannot be easily enhanced to model the features required by a distributed real-time system. This is because the existing models do not explicitly model the communication system or include timing information. Therefore a requirement exists to develop a new modelling and analysis technique to support the development of such protocols.

The implementation of existing protocols is examined in section (2.6), a major factor being the type of communications used. Typically, asynchronous local area networks have been used as the communication system but no guarantee on timely transmission and message ordering can be provided. Thus an implementation that can guarantee such properties is required if the protocols are to be used in distributed real-time systems.

The final aim of this chapter is to demonstrate that the protocols developed can be used in distributed real-time systems. Section (2.7) assesses the need for such protocols in distributed real-time databases and distributed real-time control systems. To demonstrate the usefulness of the protocols it is preferable that they are modelled using a technique which can also model the application. This allows a uniform modelling and analysis technique to be used throughout the thesis.

2.2. The coordination of distributed systems

Coordination is needed in distributed systems to prevent concurrent operations from having undue effect. Two operations may be executed alone without any problems but if they are executed concurrently they may need to be coordinated so that one finishes before the other starts, this is known as serialization. A similar coordination problem occurs if two operations access the same resource, the operations must be coordinated so that only one accesses the resource at any time. This is known as mutual exclusion and is similar to serialization but is used when it is not known which operation should occur first. Atomicity is another form of coordination which requires that either a number of events complete satisfactorily or they all have no effect.

In a distributed database system, user operations are performed as transactions (see section (1.2)). Since user operations can be initiated concurrently (either at the same site or distributed) transactions must be executed in a manner that prevents interference. Also, since failures can occur, transactions must ensure that either all the operations of a transaction take place or the transaction has no effect on the database.

A transaction executed alone will always transform the database from one correct state to another. Therefore two transactions executed serially will also result in a correct database. To prevent concurrent transactions interfering, concurrency control techniques are used. If two transactions are executed concurrently at the same site then their interleaved execution must produce the same result as if they had been executed serially. This also applies when the transactions are executed at different sites. The interleaving does not have to be a serial execution of the two transactions thus allowing greater concurrency than serialization. If the interleaving of two transactions does produce the same result as a serial execution then the execution of the two transactions is said to be serializable.

A correct database can be maintained by assigning integrity checks on database items but even for small transactions the number of integrity checks would be large. To prevent such a large number of integrity checks serializability can be used as the consistency constraint. If two transactions are not serializable then they cannot be executed concurrently.

The most common example of how concurrent transactions interfere is the lost update problem. As an example of how updates can be lost consider transactions T_1 and T_2; T1 increments the data item X by 10 and T_2 decrements X by 5. If initially X is 20 and T_1 and T_2 both occur serially then the expected result would be X = 25. If T_1 and T_2 are executed concurrently then the operations on X could be :-

(1) T_1 reads X = 20

(2) T_2 reads X = 20

(3) T_1 operates, X = 30

(4) T_2 operates, X = 15

The final result is either X = 15 or X = 30 depending on which of T_1 or T_2 terminated last. To control such problems database systems use concurrency control methods to preserve serializabilty of transactions.

Concurrency control is a well developed field both for centralized and distributed systems [Bernstein 87a]. Many different methods exist but most are variations on one, that is 2-phase locking [Eswaren 76]. Other concurrency control mechanisms can be categorized as either being timestamp methods [Bernstein 80] or optimistic methods [Bhargava 83, Kung 81].

The accepted standard concurrency control solution, 2-phase locking, is inefficient and reduces concurrency considerably but has the advantage of being simple to implement. 2-phase locking involves locking a data item when a transaction requires it and making other transactions that need to access the data item wait until it is free again. This prevents conflicts between concurrent transactions. A transaction must acquire a lock on a data item before it can access it. The basic idea behind 2-phase locking is that once a transaction has released a lock it cannot acquire any new ones. This means a transaction obtains locks in two phases :-

Phase (1) Locks on data items are accumulated

Phase (2) Locks on data items are released

Two types of locks are used, a read-lock is used when reading data and a write-lock when updating data. Since transactions may be concurrent, two transactions may attempt to lock the same data item. Locks on the same data conflict in two ways, a read-write conflict and a write-write conflict. The first occurs when a transaction holds a read lock and another attempts a write lock on the same item. A write-write conflict occurs if a transaction holds a write lock and another attempts a write lock on the same data. If the locks conflict then access is denied to the latter transaction.

Using this technique serializability is ensured and the consistency property of a transaction is performed. To provide the isolation property of a transaction the locks must be held until the transaction is terminated upon which the locks are released instantaneously. This is because otherwise exclusive locks could be released at any time during phase (2) and could be read by another transaction even though the result may be aborted later.

Other non 2 phase locking protocols exist and allow more concurrency than 2 phase protocols. Even greater concurrency can be achieved using these protocols by allowing the type of locks held to be converted dynamically [Mohan 85].

Another method for concurrency control is to assign ordering to database accesses before execution of the transaction. Each transaction is given a unique identifier, the timestamp, which is tagged to all read and write operations. Each data item is now associated with the largest timestamps of the possible read and write operations. Conflicts are then detected by checking the timestamp of an operation against the stored largest timestamps. For a read, if its timestamp is less than the largest write timestamp it is rejected and the corresponding transaction is restarted. Similarly for a write operation, if its timestamp is less than the largest read or write timestamps then the transaction is rejected and restarted. If there is no conflict the operation is allowed to continue and the corresponding largest timestamp for the data is updated to the timestamp of the operation. One problem with using timestamps in a distributed system is the difficulty in maintaining global clocks [Lamport 78] but this is usually solved by using a site identifier tag.

Unlike 2-phase locking and timestamping, optimistic concurrency control methods execute a transaction until completion [Kung 81]. To prevent conflicts all write operations are performed on local copies. At the end of the transaction global validation is applied, if this is passed the database is updated, if it fails the transaction is restarted. The validation test ensures that the execution of the transaction is serializable and is usually based on timestamps. This method is optimistic in that it assumes more transactions pass the test than fail.

2.3. Errors and faults

A failure of a system is said to occur when its behaviour deviates from that specified [Randell 78]. A failure stems from the system containing an uncorrected erroneous state. Erroneous system states are caused by faults, these faults can either be component faults (hardware) or design faults (software).

Faults in a distributed computing system can be attributed to either hardware or software. Hardware faults are mainly due to the deterioration of components as they age. Software does not age and so faults are due to errors introduced at the design stage. Hardware failures can be expected to happen and systems designed to minimize their effects. Software faults are unexpected because the design errors introduced cannot be predicted.

Since it is impossible to prevent hardware from deterioration, hardware failures must be tolerated. Techniques to tolerate hardware faults consist of replication [Lala 85]. This redundancy is usually provided either dynamically or statically. When a number of identical hardware elements are executed concurrently it is known as N-modular redundancy, this is known as static redundancy. When a result is required the outputs from each of the units are passed to a voter unit, if a unit has failed its failure is masked by the results from the other units. Another method to tolerate a hardware failure is to use dynamic redundancy in the form of a standby unit which repeats the processing required after a failure of the original unit is detected. If a standby unit can be switched in and finish processing within a known deadline it is possible for this technique to be used in real-time systems.

In a distributed computing system there are a number of possible sources of hardware failures, sites may fail and communication errors may occur. This thesis assumes that when a site fails it stops completely and does not do any erroneous processing, this is known as the fail stop approach [Schneider 87]. An alternative would be to allow sites to fail unpredictably, eg to compute strange values or send conflicting information to different parts of a system. Detecting such failures and dealing with them is very complex and is known as the Byzantine generals problem [Lamport 82]. In a distributed database system if a site fails, other sites can only perform transactions that access data at the failed site if the data is replicated at a working site in which case the failure has little effect. In a distributed control system, failure of a site can be disastrous because control over an actuator is lost. For example a site controlling a valve may fail with the valve open. The only solution is to use backup processors to take over control of the valve. Without backup processors the recovery time of a failed site cannot be determined a priori and so deadlines cannot be applied if sites are assumed unreliable. Thus real-time control systems cannot be made resilient to site failures without using redundant processors.

In general the communication network may fail in a number of ways depending on the type of communications used. Messages may be delivered out of order or late and if failures occur messages may even be lost. If a direct point-to-point network is used the order of messages is preserved and the propagation delay of messages is known. This thesis investigates the use of both asynchronous and synchronous point-to-point networks in real-time distributed control systems. One particular feature investigated is their resilience to unpredictable disconnection.

Software is vastly more complicated than hardware and is also very application specific. This increases the probability of design errors being undiscovered even if rigorous software engineering techniques are used. Two techniques exist for reducing the effect of software faults, fault prevention and fault tolerance. Fault prevention attempts to reduce the number of design errors during development, whilst fault tolerant techniques are used to mask faults that do occur.

The use of formal methods is becoming an ever popular approach to fault prevention. Formal methods are based on a mathematical or logical foundation and therefore have a set of rules which allow reasoning about a system. Formal methods consist of two approaches, formal specification and formal verification. Formal specification [Parnas 77] is the precise representation of a problem using a mathematical representation which has a predefined set of rules. There are a number of specification tools, many such as Z [Spivey 89] are based upon set theory and predicate calculus. Another form of formal specification is the use of graphical techniques, these are known as formal models. For example formal models such as finite state automata and Petri nets can be executed to determine the outcome of the system. Formal verification [London 75, Keller 76] is used to ensure a program adheres to its specification. London [London 75] suggests that verification of a program should not be used as a substitute but in conjunction to testing.

The use of formal methods is slowly filtering through into concurrent systems [Lamport 83] and real-time systems [Joseph 89] but their use is still difficult and error prone. From the experience of Moser and Melliar-Smith [Moser 90] it is clear that formal verification is only possible for small code segments. Undoubtedly progress in this area of research will be rapid. A discussion of these methods is beyond the scope of this thesis but a survey detailing common techniques for real-time systems including concurrent systems can be found in Sagoo and Holding [Sagoo 90].

An informal technique which can be used to eliminate hazards is known as software fault tree analysis (SFTA) [Leveson 83]. A design is analysed for potential hazards, the causes of which are identified until primitive faults are reached. The design is then altered to prevent the primitive faults. The reliability of a system cannot be estimated using this method unless the probability of the causes are known, this is difficult (if not impossible) for design faults and so achieving a desired level of safety is unlikely.

Software can still contain design errors even after rigorous development. To mask such faults software fault tolerance techniques must be used [Anderson 81]. Such techniques are described in the next section and code redundancy.

2.4. Software fault tolerance

Due to the immaturity of formal methods it is unlikely that design errors can be completely eliminated from software therefore techniques to mask faults caused by these errors should be used. Such software fault tolerant techniques mask faults by using redundant code. All of the fault tolerant techniques described here work on the principles of error detection, damage assessment and error correction. The final stage of a general purpose fault tolerant mechanism, fault removal is not used in software because code is not dynamically reconfigurable.

2.4.1. Traditional software fault tolerance methods

Software fault tolerance methods involve either static or dynamic redundancy to mask faults. Static redundancy executes redundant code even when failures do not occur whereas dynamic redundancy executes redundant code only when necessary. Thus dynamic redundancy involves less overheads. The basic assumption behind the use of redundancy in software is that different designs from the same specification will contain different errors.

N-version programming [Avizienis 85] is an example of static redundancy. Here N-versions of a program are executed in parallel and their outputs passed to a voting mechanism. Majority voting is then performed using all the program outputs, thus if a majority of outputs agree erroneous results can be masked.

A problem with this technique is that it assumes that design diversity will produce different design errors. Knight and Leveson [Knight 86] carried out a statistical study on 27 independently written, identically specified programs. The programs were subjected to many different sets of data and any faults were monitored. This study showed that common design errors do occur in independently developed programs, probably due to a common misinterpretation of the specification. This illustrates the need for formal specification methods. Another problem that may also occur is in the use of floating point arithmetic, results from different programs may be correct but may not match due to rounding errors [Brilliant 89]. For these reasons N-version programming should be used carefully, probably in conjunction with other software fault tolerant techniques.

Dynamic redundancy does not incur the overhead of executing redundant code unless an error is detected. Instead when an error is detected an attempt to re-do the faulty computation by using an alternative piece of code is tried. Before the alternative can be executed an error free system state must be re-established, this is achieved by using either backward or forward error recovery.

Recovery blocks [Randell 75] use dynamic redundancy and are used in sequential systems to provide backward error recovery. A recovery block consists of a recovery point, an acceptance test, a primary module and a number of alternative modules. When the recovery block is first entered the error free system state is stored, this is known as the recovery point. The primary module is then executed. When this is completed the validity of the results are checked by an acceptance test. If the acceptance test is passed, the recovery block is exited and the saved recovery point is discarded. If the acceptance test fails then the program is rolled back to its recovery point, the saved state is restored and an alternative module is executed. This continues until the acceptance test is either passed or the number of alternatives are exhausted. If the acceptance test fails and no more alternatives exist then the recovery block fails and the system must provide recovery at a higher level, possibly by nesting recovery blocks.

This technique does not incur the overheads of N-version programming if faults do not occur but it does suffer from the same design diversity problem. Another problem with recovery blocks is that the acceptance test is a critical component. It must be designed to detect as many errors as possible but also be minimized because it is executed every traversal of the recovery block. Acceptance tests are devised to detect unexpected program executions and also prevent unsafe output, Hecht [Hecht 79] describes how acceptance tests can be constructed.

Fault tolerance in a concurrent system where processes communicate is much more difficult to achieve because errors can migrate during inter-process communication. Consider a number of communicating parallel processes. If an error is detected after a communication has taken place, the communication must also be undone during rollback . This means the process receiving the message must also be rolled back, this process may also have communicated with another process and so that communication must also be undone. This cascading of rollbacks is known as the domino effect [Randell 75] and may lead to all processes being rolled back to their start. A method of controlling error migration and recovery is therefore needed.

Atomic actions are a method of controlling error migration by ensuring that the processes within an atomic action do not communicate with processes outside for the duration of the atomic action [Anderson 81]. When an error is detected each process is rolled back to the state occupied when entering the atomic action. Since no process communicates outside the atomic action the domino effect is limited. Atomic actions can be associated with backward and forward error recovery techniques.

The most well known technique using the ideas of atomic actions and backward error recovery is the conversation [Randell 75]. A conversation involves the coordination of recovery for a number of concurrent processes. It is basically an extension of the recovery block mechanism which limits inter-process communication. A conversation is often described as a boundary consisting of a recovery line, an acceptance test and sidewalls to prevent communication. The recovery line consists of a set of process recovery points, one for each constituent process and is used during rollback. The acceptance test of a conversation is a test on the set of results produced by the constituent processes, no process is allowed to leave a conversation unless all pass the acceptance test. If any fail then all the processes are rolled back to their recovery point and the conversation is restarted using alternative modules (for the same processes).

A major problem with using conversations is the identification of constituent processes and a suitable boundary. Tyrrell and Holding [Tyrrell 86] describe a static method using process states to solve such problems. This method is simplified by Wu and Fernandez [Wu 89] who only consider interprocess communication as being important. This simplified method loses state information and only provides an approximate boundary. One problem with using such a method is that conversation placement takes place after the system is designed and transformed into a state representation. An alternative solution would be to identify where conversations are required during the design stage.

Other problems of the static design of conversations exist because once a set of processes are chosen to take part they cannot be changed. Alternative approaches have considered the dynamic design of conversations [Merlin 78, Kim 88], although great care is required to avoid the creation of recovery modes that exhibit the domino effect. Problems also arise from the generic structure of the conversation. Processes enter a conversation asynchronously but are synchronized on their exit because of the acceptance test. This implies that if a process never enters a conversation then all the other processes cannot leave, the conversation does not allow any timeout mechanism. A problem also occurs when the acceptance test fails and the processes are rolled back and execute their alternatives. It may be better for just a few to execute their alternatives or for a completely different set of processes to interact. Also because primary modules are different to alternative modules a new acceptance test may be preferable to the original.

These problems are solved by Gregory and Knight [Gregory 85] who proposed a linguistic approach known as the colloquy. Again a colloquy uses atomic actions and backward error recovery but unlike conversations different processes are allowed after recovery. A colloquy consists of a number of blocks known as dialogs. A dialog indicates the processes that take part in an atomic action and the acceptance test to be used. If a dialog fails its acceptance test then the colloquy can execute another dialog. This method also allows timeouts to be used. An attempt was made to provide such a mechanism using a production programming language [Gregory 89] but found the differences between the needs of a general purpose programming language and backward error recovery difficult to overcome. They also suggested that a programming language with semantics designed to support backward error recovery is needed.

The previously discussed techniques, the conversation and colloquy provide structuring of backward error recovery using atomic actions. However, in certain systems forward error recovery is more useful [Leveson 83]. Atomic actions can also be used to structure forward error recovery [Campbell 86, Taylor 86]. Problems occur when concurrent processes detect different faults because they may be handled individually instead of as one. Campbell and Randell [Campbell 86] propose the use of an exception tree to eliminate this and activate the required fault tolerant measure.

All of the previously mentioned techniques are useful for tolerating software faults and can be used to tolerate certain transient hardware faults, e.g. a memory bit changing.

2.4.2. Providing atomicity in the presence of faults

None of the fault tolerant mechanisms described above provide atomicity, although the acceptance test of a conversation does provide a fail/succeed mechanism there is no guarantee of this if all the alternatives fail. A database system uses transactions to provide atomicity in the presence of hardware failures and concurrent transactions. Atomicity when concerning transactions is often divided into failure atomicity and concurrency atomicity. Failure atomicity means that the transaction either completes successfully or has no effect on the database, this is the usual meaning of atomicity. Concurrency atomicity of a transaction is the property that partial results are not available to other transactions. This thesis uses the term atomicity to mean failure atomicity and concurrency control to mean concurrency atomicity. The mechanism used by transactions to provide atomicity is extended with timing requirements in this thesis. Its use in real-time systems is then considered.

Stankovic [Stankovic 88b] suggested that real-time databases could take advantage of cut-down transactions to satisfy timing constraints. This thesis takes the idea of cut-down transactions further and just uses the mechanism which provides failure atomicity. Such mechanisms are known as commit protocols. The basic idea of a commit protocol is to ensure the all/nothing property even when failures occur, this ensures that a consistent state is always maintained. Commit protocols are surveyed in chapter 3

An example of the use of a commit protocol in a database system is the termination of a transaction which has spawned a number of sub-transactions. To maintain a consistent database all of the sub-transactions must be terminated in the same way. Either they are all completed or they all have no effect. This is provided by the commit protocol which ensures that either all the sub-transactions are committed or none of them are. A commit protocol uses stable storage (eg. disk) to store system states for use during recovery after a failure. Stable storage is assumed to be tolerant to failures in that memory contents are never lost.

Initially the acceptance test of a conversation appears to provide the all/nothing property required for atomicity because either all processes exit or are restarted. The difference being that the processes are not just rolled back but restarted using an alternative module. If a number of faults occur then this roll back will be repeated until there are no more alternative modules and the conversation is deemed to have failed, recovery must then be performed at a higher level. Thus there is no guarantee on the overall outcome. Also processes enter a conversation asynchronously therefore some may enter before it is known that all the processes are able to start. A commit protocol overcomes this by ensuring that if a constituent process cannot take part then no processes take part. Another problem with the conversation is that although they tolerate software failures they do not tolerate hardware failures because they do not use stable storage. Commit protocols on the other hand are resilient to any number of sequential failures and use stable storage to store system state. Conversations are designed for process interactions whilst commit protocols are expected to operate with data items.

Since a safety critical system needs to tolerate failures and still remain safe, the coordination of such a system requires extra care. If the system is distributed the coordination of the system is even more difficult because inter-processor communication failures must also be tolerated. Not only must software faults be tolerated but also hardware faults. The above mentioned fault tolerance techniques can be used to tolerate software faults but not hardware failures. A protocol used to coordinate safety critical systems must be able to function correctly but always maintain a safe system even in the presence of hardware failures. If a failure occurs which prevents the desired operation being performed within its timing constraints, a substitute operation which meets these constraints can be used.

This thesis proposes the use of commit protocols to coordinate processes in real-time systems. The coordination is such that if a timing constraint cannot be met then a safe system state is still maintained. One problem of commit protocols is that the all/nothing property is always reached eventually, no timing constraint is applied. Therefore this thesis also considers the design of commit protocols enhanced with timeout mechanisms to provide timely decisions.

2.5. Design of robust commit protocols

The preferred characteristics for real-time commit protocols have been identified above. To satisfy these requirements it is desirable to develop a technique and model which can be applied to the specification, design, implementation and verification of the protocols. A protocol for use in real-time systems must provide the required functionality and also meet pre-determined temporal constraints. This thesis assumes that the only temporal constraint on a commit protocol is that the decision must be made before its deadline. The functional properties that a commit protocol must satisfy are that the outcome at each site is consistent with the other sites and also correct.

A system model which shows the state of each site in a commit protocol can be used to demonstrate the desired functionality. The commit protocols can be modelled as a series of state transitions at each site which in the absence of any failures should produce a consistent result. Failures can then be included in the model and the design enhanced to provide resiliency to such failures. The scope of failures included depends on the aspects of the system modelled.

Only one formal model for commit protocols currently exists, this is described in the next section. This thesis examines the limitations of this model and proposes a new model which can be used as a basis for the design of robust protocols.

2.5.1. Previous modelling of commit protocols

The first formal model of commit protocols used a finite state machine (FSM) to model local states [Skeen 83]. Each site was modelled by an individual FSM and the global system state was defined as the union of all local states and any outstanding messages in the network. The analysis of this method is explained in more detail in chapter 3.

Skeen's model includes implicit assumptions about the communication network. These assumptions simplify the model but limit its usefulness. For example communication mechanisms are not modelled with the result that the model of communication failures is not explicit. Typically, the model assumes that timeout messages are generated by the network when a message fails to be transmitted. Another assumption made is that the transmission of a number of messages is atomic. This means that if a site sends a message to a number of other sites then if one site receives the message it is assumed that all the other sites receive theirs. Obviously in a practical distributed system this situation is unrealistic.

The original model does not contain any timing information and so cannot be used to derive performance characteristics, without which it cannot be deduced if deadlines are met. This was not a problem in Skeen's work [Skeen 83] because the commit protocols did not have associated timing constraints. However it is a severe limitation if the protocol is to be used in real-time systems.

The FSM models were used to derive a number of theorems about the existence of commit protocols when various spurious faults were allowed. One of the most significant results that have been obtained is a theorem which states that there does not exist any commit protocol using independent recovery which is resilient to arbitrary failures by more than one site [Skeen 83], thus the resiliency of commit protocols is limited.

Skeen's assumption that multi-site communications constitute an atomic action is relaxed by Yuan [Yuan 89]. This extends Skeen's model so that a site can fail after sending a message to only some of the intended sites. This is much more realistic for a multi-site system. Yuan also proposed a new protocol which is resilient to multiple failures but only by communicating with other sites after recovery. Thus upon recovery a site knows what action all the other sites took and can act appropriately. The protocol is enhanced so that failures which allow independent recovery can be detected and used for optimizing the number of messages sent. This reduces the communication overhead and therefore the recovery time. One disadvantage is that a fully connected commit network is required.

2.5.2. Petri net modelling of commit protocols

Petri nets [Peterson 81, Murata 89] are a technique for modelling sequential and concurrent systems. They are mathematically based and can be represented either as a set of equations or as a bipartite directed multigraph. The graphical representation is clear and precise and allows the dynamic and concurrent properties of a system to be visualized. Petri nets model control flow and can show synchronization and communication between parallel activities. A FSM is a subclass of Petri nets known as state machines [Peterson 81] but do not show such synchronization or communication.

Many uses exist for Petri nets, in particular they can be used to model computer hardware [Azema 76] and distributed software systems [Mekly 80]. Another use is as a high level design tool, Nelson et al [Nelson 83] have suggested that by annotating Petri nets they can be transformed into a suitable programming language. Another recent use of Petri nets is their use in designing software fault tolerant systems [Tyrrell 86, Leveson 87].

The analysis of Petri nets allows system properties such as liveness and reachability to be verified. Liveness shows that a deadlock cannot occur and reachability shows if a possible global state is achievable. Two techniques exist for verifying such properties, the transition matrix and the reachability tree [Peterson 81]. Although more amenable to computer analysis the transition matrix does not represent state as clearly as the reachability tree. The reachability tree of a concurrent system can be very large even for a system with few states, the analysis of which is often automated [Razouk 85b, Memmi 84].

This thesis considers the use of Petri nets to design robust protocols for use in distributed real-time systems. The Petri net are shown to be correct by the manual generation and analysis of the reachability tree. Petri nets and their analysis are described in chapter 4. It is then shown in chapter 5 how the Petri net can be transformed almost directly into the programming language Occam.

Petri nets can be used to model the states of a concurrent system, thus they can be used to model the state based behaviour of commit protocols. They can also be used to model the same type of failures as Skeen's FSM model, i.e. site and communication failures. In addition they can be used to provide a uniform method of modelling failures. Following Leveson and Stolzy [Leveson 87] who allow failures to be incorporated by using failure transitions, the same method is used in this thesis to model site failures. Additionally, lost messages are modelled as failure transitions and communication failures as unfireable transitions.

Using Petri nets a succinct representation of the communication system can be developed. This allows individual messages to be modelled. Unlike the FSM models, which did not include explicit messages, communication failures can now be added to the Petri net model. This allows recovery mechanisms to be placed appropriately. The FSM models are provided in chapter 3 whilst the Petri net models are derived in chapter 4. Modelling communications allows a comparison between the use of asynchronous and synchronous communications and it is also possible to identify where optimization may be used.

Another important feature of Petri nets is that time can be included, this allows analysing of the net with respect to timing properties. Properties such as the determination and placement of timeouts can be found by analysing the timed Petri net. Such timeouts can be used to ensure that a consistent decision is always made before the deadline.

2.6. Implementation of commit protocols

Commit protocols have traditionally been used in distributed database systems where the only means of communications is by a local area network [Ceri 87]. An early distributed database system, SDD-1, used a communication network designed to provide each site with facilities for reliable communications [Hammer 80].This network is known as Relnet (reliable network) and provides guaranteed message delivery in the correct order. If a site fails and a message cannot be delivered spoolers are used to store this message until the failed site recovers. The commit protocol used by this system is detailed in chapter 3.

The underlying communication structure affects the implementation of commit protocols because it restricts how sites can interact. For instance commit protocols can be designed to communicate using centralized, hierarchical or linear communication structures [Ceri 87]. A centralized communication structure designates one master site which can communicate with every other site but these other sites cannot communicate with each other directly only via the master. However a hierarchical structure allows messages to be passed up and down the tree and for sites within a local sub-tree of the hierarchy they do not need to go via the master site. The communication structure therefore affects the design and implementation of the commit protocols.

In this thesis only the more commonly used centralized commit protocols are implemented because they can be easily modified to use other communication structures. The language Occam [Inmos 88a] is used to implement the commit protocols because of its succinct message passing ability and because each primitive in Occam has a corresponding Petri net model [Carpenter 88a]. Since Occam uses direct point-to-point synchronous communications, any facilities for reliable communications previously provided by the local area network must now be provided by the commit protocol itself, an example being the provision of timeouts. However, such protocols must be proved to be free from deadlock, inconsistencies, and timing errors for the given communication structure.

In chapter 4 Petri net models are used to demonstrate that commit protocols can be designed using direct synchronous communications and still be resilient to failures. It is then demonstrated in chapter 5 that the Petri net specifications can be transformed into the programming language Occam. The proposed implementation also identifies a known short coming of the Occam language which prevents direct implementation when communication failures are allowed, this is overcome by using a low level extension to Occam.

2.7. Application of commit protocols to real time systems

The applications of the protocols developed are demonstrated in chapter 6 by three examples. Firstly a real-time database example illustrates how commit protocols with deadlines may prevent transactions missing deadlines. The next two examples illustrate the use of commit protocols in control systems.

2.7.1. Applications in real-time database systems

A real-time database system is usually very closely related to its environment, possibly as part of a control system, this implies that access time to the database must be fast and predictable [Singhal 88]. This response time must be maintained even in the presence of failures whenever possible. The response time of a database can be improved by using larger main memory (less disk accessing), trading a feature (such as serializability) or by replicating data.

Data replication is often used because it is less costly to achieve than main memory systems and the full properties of a transaction can be provided if necessary. Another advantage is that it is unlikely that all the copies of a data item will be unavailable due to failures and so transactions can still execute [Son 87]. If critical data is replicated at sites which access it frequently then the response time of transactions accessing this data can be reduced because communication overheads are not incurred.

Problems with replicated databases are the storage overheads and the control of replicated copies. Replicated copies must be made to act as a single unit, ie a strong consistency constraint is needed to ensure all copies of a data item always have the same value. This is impossible to fulfil all the time because it takes finite time to update data and communicate requests. This can be temporarily relaxed but a mechanism is needed to resolve inconsistencies as soon as possible [Son 88, Bhargave 87].

Two conflicting properties in a real-time database are consistency and availability [Lin 88]. Whilst consistency is important it may be better to temporarily sacrifice consistency so that a required response time can be satisfied. Consider for example two copies of data item X,

x_1 and x_2 which reside at sites s_1 and s_2 respectively. If a transaction T_1 is updating both copies when site s_2 fails, to achieve consistency, x_1 cannot be updated until s_2 recovers. If another transaction T_2 now requires access to X it must also wait for s_2 to recover. A possible solution is to allow x_1 to be updated and allow T_2 to execute, x_2 is then updated when s_2 recovers. This increases the availability of X but reduces the consistency between copies because it is not known how long s_2 will be down. This causes problems because transactions can occur at any time, even when a failed site is recovering. Therefore if the item x_2 is being updated during recovery but another transaction T_3 is updating x_1 then this update is missed at site s_2. A solution to this is suggested by Bhargava [Bhargava 87] who uses the idea of a lock which indicates if an item is updated when one of its copies is unavailable.

If the situation is simplified so that transactions are not allowed during the recovery of a site then a possible solution is to use a non-blocking commit protocol. This could be used in the above example to update x_1. Since the protocol is independently recoverable, when s_2 recovers it will update itself to the value of x_1. This obviously simplifies the problem in a distributed database but is a good example of the use of such protocols.

2.7.2. Applications in real-time control systems

A distributed real-time control system coordinates the actions at independent sites by using only message passing. The coordination must also satisfy timing constraints. Often it is required that a number of sites are coordinated so that either all sites perform their actions or none of them do. It is quite reasonable to expect timing constraints to be applied to such coordination. This thesis assumes that the all/nothing property is crucial to the safety of the system, if only some actions are performed the system becomes unsafe. An example of this type of behaviour can be seen by examining again the problem of coordinating 2 robot arms which lift faulty containers from a conveyor belt. The container must be lifted by both arms, if only one robot attempts the lift the container will tip, also the lift must also be completed in time to clear the next container. This example illustrates both the all/nothing property of the actions and the timely decision to lift the container. If the decision to lift the container is late, the robots will not be able to perform the lift in time to clear the next container. Another problem that must be solved is tolerating failures, a consistent action must be performed by the robots even if failures occur.

Parallel work by Davidson et al [Davidson 89] has extended the basic two phase commit protocol with a deadline but their environment does not allow failures to be tolerated. The protocol instead allows an extra termination state, the exception state, a site enters this state when a failure occurs. This does not provide total consistency because when the protocol terminates some sites may have aborted/committed whilst others may be in the exception state, this is resolved later by a recovery procedure. The commit protocol used by Davidson is the most basic commit protocol which does not provide independent recovery. This means if a site fails the other sites must block until the failed site recovers. Since only one type of protocol is used a formal model is not used to demonstrate the required features. This work has been extended by Lee et al [Lee 89b] to take advantage of the periodic nature of many real-time tasks but uses the same basic protocol.

This thesis applies an extended two phase commit protocol enhanced to accommodate deadlines and timeouts to real-time systems and shows how site and communication failures can be tolerated. It is shown how the real-time systems can be modelled using Petri nets to show that they function correctly both functionally and temporally. The resilience of the systems to failures is demonstrated by including site and communication failures in the Petri net model.

2.8. Discussion

The use of commit protocols in real-time systems requires protocols extended with deadlines and mechanisms to detect and tolerate faults. A method of modelling such protocols and verifying their functional and timing properties is also required.

Previous formal models for commit protocols have concentrated on proving the existence of certain classes of protocols.Timing has never been included. This thesis extends the previous work by using Petri nets and including timing constraints in the specification.

The models developed in this thesis allow the communication system to be modelled thus allowing communication failures to be analysed. Modelling the communication system also allows the use of both asynchronous and synchronous communications to be compared. They also allow the incorporation of time into the models so that a uniform model is provided which makes analysing the protocol easier, timing constraints can be directly seen from the analysis of the specification.

Naturally, the protocols developed in this thesis are suitable for implementation on appropriate communication networks. The implementation of a commit protocol for use in a real-time system should provide reliable communication with a predictable propagation delay. Traditional local area networks do not provide such predictability, however the implementation of the synchronous protocol in this thesis overcomes this by using direct point-to-point synchronous communications.

When applied to real-time systems it must be shown that the commit protocol performs as expected. In this thesis, Petri net techniques are used to model both the application and the protocol. This allows the commit protocol to be modelled as an integral part of the system. From this model it can be confirmed that a safe system state is always maintained and also shows that timing constraints are met.

Chapter 3

Modelling Atomic Commit protocols

3.1. Introduction

To ensure consistent decisions are made over a distributed computer system commit protocols are used to provide atomicity. Various protocols have been designed to provide atomicity in database applications [Gray 79, Balter 81, Skeen 82a] but these have never included deadlines, the agreement is always reached eventually. This chapter describes the most important of these protocols and explains how they can be classified into blocking and non-blocking depending on the affect of a failure. The protocols are examined and investigated for their applicability to real-time systems. Non-blocking protocols are identified as being useful because they can recover independently after a failure.

It is advantageous to describe commit protocols in a precise and clear manner so that a systematic design method can be developed. The graphical model described in this chapter is the Finite State Machine (FSM). Each site is modelled as a separate FSM which expresses the local state of each site. Sites are modelled as starting in an initial state and progressing through a number of states until a final commit or abort state is reached. If all the local states are combined a global state is derived which allows the system to be analysed as a whole. A commit protocol FSM is said to be correct if all the final global states contain either all abort states or all commit states.

In many ways a formal graphical model is better than a formal mathematical model for describing distributed systems. Graphical models are usually clearer and more concise than mathematical models. Also the required information is generally found by analytical inspection rather than mathematical reasoning. For this reason, proofs deduced from a graphical model are known as rigorous and not formal. When investigating commit protocols the property of interest is that of state consistency, this can be found by inspection of the FSM model. Possible failures can be included in the model so that the resiliency of the protocol can be examined.

Existing models of commit protocols using Finite State Machines (FSM) are examined in this chapter. Previous work [Skeen 82a, Skeen 83] used the models to determine the existence of protocols which can survive various site or communication failures. Skeens results show that commit protocols can be designed to tolerate only a limited set of failure scenarios. Another result is that failure modes must be completely distinguishable. The analysis of these models provide information on what types of failures the protocols can tolerate and still be made to terminate consistently. The models are also useful to show where timeouts should be placed to provide resiliency to failures. However, the existing models are shown to be inadequate for describing timing properties or calculating the values of such timeouts.

Throughout this chapter the protocols are assumed to have access to an asynchronous communications network which fully connects all the participating sites. The network is assumed to be able to absorb undelivered messages and also report timeouts to the sender. Such a network could be a local area network which incorporates special facilities such as timeouts for undelivered messages, these facilities aid the resiliency of the protocols to failures.

These assumptions result in significant simplifications of the protocols. Later, in chapter 4, consideration is given to replacing the asynchronous communications network with synchronous point-to-point communications.

Section (3.2) describes existing protocols which have been developed in the database field. These protocols are classified as either blocking or non-blocking and their strengths and weaknesses are discussed. One specific protocol, the extended 2 phase commit, is emphasized because all operational sites can terminate consistently without waiting for the failure to be repaired and without any extra communication. Because of this the extended 2 phase commit protocol is selected for use in later parts of the thesis. The FSM models for the common protocols are discussed in section (3.3) and are analysed in section (3.4). The analysis involves generating the global state tree which consists of all possible achievable states. A method which uses this tree to determine whether a protocol is non-blocking is described. The disadvantages of the FSM model are discussed in section (3.5).

3.2. Atomic commitment protocols

Commit protocols have been used in distributed database systems to provide the atomicity function of a transaction and keep the database in an overall consistent state. They have been used to ensure that transactions either complete fully or have no effect on the database even when the database recovers after failures. Recovery in centralized databases is relatively simple but becomes more complex for distributed databases [Bernstein 83]. The added complexity for distributed databases is because the commit protocols have to ensure all the sites make a consistent decision even in the presence of failures. This thesis considers the application of commit protocols to distributed real-time systems therefore emphasis is placed on protocols applicable to distributed systems.

Commit protocols can be classed as blocking or non-blocking protocols, a method to identify blocking protocols is given by Skeen [Skeen 83]. Consistency and availability are conflicting properties for commit protocols. Blocking protocols have low availability and high consistency but non-blocking protocols have high availability but may have low consistency. Blocking and non-blocking protocols are described here with the most common protocols described in detail. Following this the concept of independent recovery is introduced and it is shown that this property is most useful for real-time systems.

3.2.1. Blocking commit protocols

A commit protocol is termed a blocking protocol if due to a site failure a consistent decision cannot be made until the failed site recovers. In a database system a blocking commit protocol reduces the availability of data because locks are held on data until the failed site recovers and the protocol terminates, thus preventing access by other transactions. The recovery time of a failed site may be quite substantial and this could be unacceptable in some situations, in particular time-critical systems. In some cases it may be better to provide a substitute answer (emergency default) within the time limit than to miss the deadline completely and possibly delay other important processes.

3.2.1.1. The two phase commit protocol

The most widely used and well known blocking commit protocol is the 2 phase commit [Gray 79]. It has been shown to be blocking by the use of a state reachability tree and the notion of a concurrency set [Skeen 83]. If a site i is in a state A, then the concurrency set for state A is the set of all possible states that other sites can occupy whilst site i is in state A. Concurrency sets are important because they allow deductions to be made about the recoverability and blocking properties of a protocol following a failure. For example, if a site fails and its state at the point of failure has a concurrency set which includes a final state then its recovery possibilities are limited (if any) because it must recover consistent with the final state of the other site. Similarly, if a protocol contains a state which has both abort and commit in its concurrency set and the site fails in this state then the protocol is unrecoverable using local information and is said to be a blocking protocol.

The 2 phase commit protocol can be centralized with one site designated the coordinator process whilst the others act as participants. The job of the coordinator is to ensure all participants (and itself if involved) make a consistent decision. The coordinator executes as follows :-

Phase 1 (1) a ready log file is written,
(2) 'ready' messages are sent to each participant,
(3) wait for participants to reply with their votes,

Phase 2 (1) when all participants have replied a decision is made ,
(2) a log file reflecting the decision is created,
(3) the decision is sent to the participants ('commit'/ 'abort').

The commit decision is only made if all the participants and the coordinator voted 'yes' otherwise the decision is to abort. The log files are used by a recovery process to see how far the protocol has got.

Each participant executes in the following manner :-

Phase 1 (1) wait for 'ready' message,
(2) if participant can commit send 'yes' else send 'no' and abort,
(3) write yes / no log file,

Phase 2 (1) either 'commit' or 'abort' message is received,
(2) write decision log file,
(3) execute decision.

Once the 'ready' message is received by a participant it checks to see if the transaction requested can be completed. If it cannot be completed the participant unilaterally aborts and terminates otherwise it continues with the protocol. In a database system a typical check would be to determine whether the data items are locked by any other transaction, if they are locked then the requested operation must be aborted and a 'no' vote is sent otherwise a 'yes' vote is sent. The second phase is necessary because another participant may not be able to commit its request therefore no others are allowed to commit. The 2 phase commit is correct with respect to participant failures but is blocking for coordinator failures. The 2 phase commit protocol described omits acknowledgement messages normally sent after a participant has written its decision log. These are omitted because they do not increase the fault tolerance of the protocol but are used to inform the coordinator that the required actions have been completed.

3.2.1.2. Variations on the 2 phase commit

This section describes a number of variations on the 2 phase commit protocol namely the presumed abort, the presumed commit and the 4 phase commit protocols. The first two are optimized versions of the 2 phase commit whilst the latter protocol includes backup processes.

The presumed abort and presumed commit protocols [Mohan 83a] reduce the number of messages and log writes involved in the original protocol and thus improve the performance. They have been implemented in the IBM distributed database system R* [Lindsay 83].

The presumed abort protocol is optimized by taking advantage of read-only transactions. If a participant receives a 'ready' message and decides no updates are needed then it sends a read vote and releases its locks on any data items. No log records are needed and no command message needs to be sent, this is because the outcome of the transaction is irrelevant to read-only participants as no updates are done. If all participants reply with read votes then there is not even a need for the second phase. If at least one yes vote is received then the coordinator acts as for the 2 phase commit .

The presumed abort protocol also attempts to overcome the blocking which occurs in the 2 phase commit protocol following failure of the coordinator. It also reduces the number of log records required. Specifically, in the 2 phase commit protocol, if a failure of the coordinator occurs after sending the 'ready' messages and before receiving all the votes then there is no information about the outcome of the decision. When the coordinator recovers, its recovery process may get inquiries from the participants which cannot be answered. In the presumed abort case if there is no information the transaction is assumed to be aborted and an abort message is returned. Thus the coordinator does not need to write an abort decision log because any recovery procedure finding no log assumes the decision is abort. This protocol still blocks when a coordinator fails before informing the participants about the decision.

An extension to the presumed abort uses Byzantine agreement [Mohan 83b] to trade more message passing for an improved recovery time. This extension uses the notion of a cluster of processes, such a cluster being highly connected with a high communication bandwidth. Byzantine agreement [Lamport 82] is used between the processors within a cluster but not between clusters because the inter-cluster communication bandwidth and connectivity is assumed to be low. The commit protocol is non-blocking if the transaction is executed entirely within a cluster because each processor within the cluster acts upon the same database. Since failures should be a rare event the extra message overhead and additional processors required are probably unacceptable for most applications.

Another extension to the presumed abort is that used by the Quicksilver system [Haskin 88] where additional possibilities exist for the vote returned by a participant. The different possibilities allow varying degrees of synchronization and recoverability.

The presumed commit protocol is the dual of the presumed abort and was designed because more transactions are expected to commit than abort. In this protocol when a site recovers and finds no information about the decision it is assumed to be commit. An inconsistency can occur if the coordinator failed after sending 'ready' messages and before collecting all the votes. Since no log record would have been written the coordinator would recover and abort because it has no record of the participants involved, the participants upon recovery would find no information and presume a commit decision. This inconsistency can be removed by including a prepared log which identifies the participants involved.

The presumed commit is more efficient for update transactions whereas the presumed abort is more efficient for read-only transactions. Intuitively more reads than updates are performed per transactions so presumed abort is preferred. Within R*, the type of commit protocol can be varied for the type of transaction but the choice for a mixed transaction must be made heuristically.

The 4 phase commit protocol has been used in the SDD-1 system [Hammer 80] and the DDM system [Chan 83] and is based upon the 2 phase commit with the addition of backup processes and the idea of a reliable network. In SDD-1 the network, Relnet has a guaranteed delivery layer which ensures that if a message is sent it will eventually be received by the correct site in the correct order. The Relnet is provided by a number of reliable buffers called spoolers. The use of such a network allows the commitment of a transaction even if a participant fails, this is because when the participant recovers it will receive the final commit message from the network. To provide resilience to coordinator failure a number of backup coordinators are assigned. The four phases of the protocol are :-

phase 1	a number of ordered backup coordinators are created,
phase 2	updates are sent to the participants,
phase 3	commit/abort is sent to all backups,
phase 4	commit/abort is sent to all participants.

If the coordinator fails one of the backup processes takes over, a two phase procedure is used to ensure all the other backups are in the same state. This is needed because a backup may fail and another one must be chosen which must know the same outcome as the failed backup.

The protocol is blocking only if the coordinator and all the backups fail at some time during the four phases, this becomes very unlikely when more backups are used. The overheads involved in creating and coordinating backup processes are very costly in terms of message transfers and are probably unacceptable for many systems.

To summarize this section on blocking protocols the application of each is considered. Firstly the 2 phase commit has the advantage of being simple and general purpose, improvements to the efficiency are provided by the presumed abort and presumed commit. The latter protocols may be useful in real-time systems because of the optimizations but they still block when failures occur. Another problem is that it may not be possible to unroll incorrect presumed commits if they involve physical output. It must be noted that the agreement of these protocols is not bounded by any time limits which would be essential for the application to real-time systems. The 4 phase commit is not considered in the thesis because of the added complexity of backups and the idea of guaranteed delivery which would be difficult to provide

3.2.2. Non-blocking commit protocols

A commit protocol is non-blocking if it can be terminated without waiting for the failed site to recover. The properties of non-blocking protocols have been investigated by Skeen [Skeen 81a, Skeen 82a] by the use of FSM's, this technique is discussed in section 3.3. Non-blocking protocols have high data availability because data is never locked indefinitely, but data may become temporarily inconsistent until a failed site recovers. This inconsistency is only between operational and failed sites, all operational sites will be consistent. The application of non-blocking protocols to real-time systems appears more feasible than blocking protocols because all operational sites can be made to terminate within a deadline. One problem with their use is that they incur approximately fifty percent more communication overheads than their corresponding blocking protocol [Dwork 83].

3.2.2.1. The 3 phase commit protocol

This protocol is very similar to the 2 phase commit but buffer states are added before entering the commit states. There exists at least two 3 phase protocols, the one presented by Skeen [Skeen 83] adds a buffer state only to the coordinating site but in general a buffer state is added to the coordinator and the participants [Skeen 81a]. This latter version allows the protocol to be the same for the fully decentralized case where each site takes part as an equal. This thesis assumes the centralized version of the protocol with buffered coordinator and participant when referring to the 3 phase commit. In addition, the simple protocol with a buffered coordinator is termed the extended 2 phase commit.

The execution of the coordinator process for the 3 phase commit protocol with reference to messages and states only is as follows :-

Phase 1 as for the 2 phase commit a 'ready' message is sent to each participant, the coordinator enters a wait state.

Phase 2 votes are collected, if any vote is 'no' then an 'abort' command is sent else a 'prepare to commit' message is sent. The coordinator enters a prepared state.

Phase 3 the coordinator collects acknowledgements from the participants and sends the 'commit' message and commits the transaction.

The participant follows a similar procedure :-

Phase 1 as for the 2 phase commit either a 'yes'/'no' is returned to the coordinator after a 'ready' is received. If 'no' is sent the participant unilaterally aborts the transaction otherwise it enters the ready state.

Phase 2 the participant waits for the command either 'abort' or ' prepare to commit'. If the later is received then a prepared state is entered else the transaction is aborted. An acknowledgement of this transition is sent to the coordinator.

Phase 3 a final 'abort'/'commit' command is received to finish the participant.

The coordinator in the extended 2 phase commit executes as above except 'commit' messages replace the 'prepare to commit' messages because the participants do not have a prepared state. Also the participants of the extended 2 phase commit are the same as the participants used in the 2 phase commit with an extra acknowledgement message.

The centralized 3 phase commit involves an extra 2N messages [Dwork 83] but this overhead is more than outweighed by the non-blocking property. Recall that the 2 phase commit is blocking if the coordinator fails and no operational site has reached their commit state, ie all sites are in their ready states. Now for the 3 phase commit if none of the operational sites are in the prepared to commit state the failed site could not have possibly committed itself therefore the transaction can be aborted. If a failure of the coordinator happens in the final phase a new coordinator can be elected to terminate the transaction because the participants must be in the prepared state and could not have aborted.

3.2.2.2. Termination protocols

When a site failure is detected a commit protocol cannot terminate normally, therefore a termination protocol is used so that all operational sites finish consistently [Skeen 81b]. Also the failed site must terminate consistently upon recovery. This would be a useful feature for real-time systems because the operational sites could be used to override the failed site but still provide a consistent decision.

Termination protocols can only be used with non-blocking protocols such as the 3 phase commit. When the failure of a participant is detected the operational sites can all terminate correctly and the failed site will terminate after recovery. As for the 2 phase commit problems occur when the coordinator fails. Upon detecting a coordinator failure a new coordinator is elected. This can be done in various ways but a simple and efficient way is that used by the SDD-1 system [Hammer 80]. Once elected the new coordinator takes over and polls each site for state information.

If any site is found to be in the commit or abort state then the outcome of the transaction is known and all operational sites must be terminated appropriately. However, if there are no sites in a final state the termination protocol can decide whether to commit or abort the transaction. This decision is based upon two principles [Skeen 81b], the first is that if at least one site is in the prepared state then the transaction can be safely committed. This is because no site can have aborted otherwise a prepared state would not have been entered. The second principle is that if at least one site is not in the prepared state then the transaction can be safely aborted because there can be no site in the final commit state. The two conditions are not mutually exclusive and often both can hold together, in which case the termination protocol has to decide on the outcome. A termination protocol that always decides to commit wherever possible is known as progressive. A non-progressive protocol will always abort when both conditions hold.

3.2.3. Independent recovery

Independent recovery is a very important property when considering protocols resilient to link failures. A link failure may occur in such a way that a site (or group of sites) become isolated, this is known as a network partition. With a network partition remote recovery information cannot be used therefore recovery must use only local information. The 2 and 3 phase commit protocols are not independently recoverable because if a partition occurred each partition would elect a new coordinator and terminate independently and possibly incorrectly.

Any protocol that uses independent recovery is very important to real-time systems because deadlines can be applied since recovery time is deterministic. In particular a protocol resilient to network partitioning is useful because each partition could still operate correctly but independently thus providing a full service.

Independent recovery has been studied by Skeen [Skeen 83] and a number of theorems and a design method have been developed. The two most important theorems arising from this work are :-

Theorem 1	"There exists no protocol resilient to a network partitioning when messages are lost ",
Theorem 2	"There exists no protocol resilient to multiple partitions".

These theorems imply that if a sending site can determine if a message has been sent or not then a protocol can be designed which is resilient to a single partitioning. Since Occam communication is synchronous, messages cannot be lost therefore there should exist a protocol written in Occam capable of tolerating a single partition. The protocols in Occam can then be directly implemented on Transputers for use in embedded real-time control systems. The ability to tolerate partitions would be extremely useful because the probability of a link failing in a real-time control system is greater than a processor failure because of the hazardous environment for links. The use of Occam for such applications therefore looks very promising because it also includes the idea of time.

3.2.3.1. Quorum based protocols

In general protocols non-blocking to partitions cannot be designed and so another technique is usually used. That is, protocols continue processing transactions when a partition occurs and merge the results after communication is repaired.

One possible answer is the primary site approach [Stonebraker 79], this assigns beforehand a site as the primary site. When a partition occurs only sites in the group with the primary site are allowed to continue processing. This method increases the likelihood of blocking but removes the need for a merge protocol. Since only one partition is allowed to process transactions response time is also increased.

Another approach is to use quorums where each site is assigned a weight (according to its importance) and a partition can only commit/abort if the appropriate quorum is formed. A quorum is formed when the number of sites willing to commit/abort agree to some predefined rules. The 3 phase commit can be extended to include quorums [Skeen 82b] by counting the acknowledgements received in the third phase.

Specifically if a network partition is detected a termination protocol is run which tries to form a quorum, if a quorum cannot be made it must wait for sites to be repaired. Once failed sites are repaired they must take part in a quorum by running a merge protocol. Quorum protocols are blocking only when a quorum cannot be made, they are also considerably more complicated than standard commit protocols and are therefore not considered further.

3.3. Finite State Machine models of commit protocols

Finite State Machines (FSM) have been used by Skeen [Skeen 83] to specify and prove existence properties of commit protocols. They are ideal for this work because of their succinct state representation This section shows how the previously described protocols can be modelled using FSM's. Since these FSM models do not include the communication system they are only suitable for consideration of systems subject to site failures. The FSM models described here are analysed in section (3.4), they are then modified to make them resilient to a single site failure. Unfortunately, FSM models lack timing information which is important in real-time systems (alternative models including timing are considered in chapter 4).

3.3.1. The 2 phase commit FSM

The 2 phase commit as described in section (3.2.1.1) can be translated into an FSM model, the two site case being shown in figure (3.1). The model shows directly the state of a site and the messages received and sent, log records are omitted for clarity but assumed to be written where applicable. The state is named within the circle whilst the labeled arcs represent messages. A message above the line is received by the appropriate site whilst the message below the line is sent to another site. Therefore a state transition may consist of receiving a message, sending a message and changing state. For more than two sites some messages, such as 'start' are sent to more than one site. The original work assumed that this was atomic, this was later relaxed by Yuan [Yuan 89]. A participant is allowed to unilaterally abort its transaction at any time before it has replied. For example it may wish to abort because it cannot acquire the required data locks or if the user aborts the transaction. The votes from the participants are collected by the coordinator whilst in state w_1. If they are all 'yes' then the decision to commit is made but if one 'no' vote is received the decision is abort. The coordinator also has a vote but this is shown as a message received in figure (3.1).

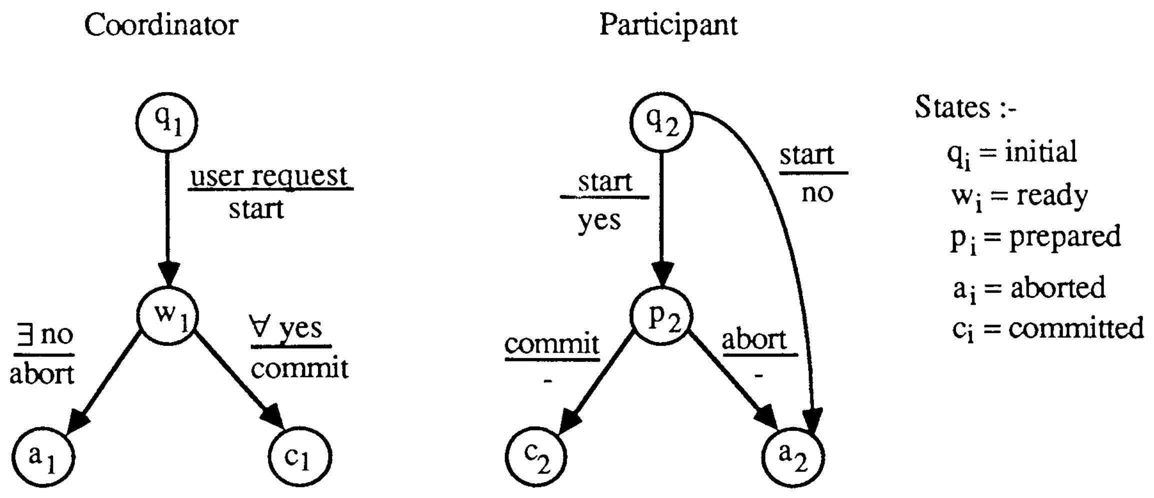


Fig.(3.1) FSM of 2 phase commit protocol

Each site progresses through a number of states until a final state is reached, if the protocol is correct then all the final local states combined should form a consistent global state. A global state can be described as a vector of local states ie. $G = (S_1,S_2, \ldots S_n)$ and to be correct the final global state vector should only contain either all abort or commit states.

3.3.2. The extended 2 phase commit

This protocol is the 2 phase commit extended with a prepared state in the coordinator and an acknowledgement message from the participant, the FSM is shown in figure (3.2).

This is an important protocol because it is an optimal non-blocking protocol [Chin 87]. It can be made independently recoverable, this has been shown to be an important property for use in a real-time system (see section 3.2.3). Section (3.4) explains why this protocol is non-blocking and how it can be enhanced so that it tolerates a single site failure.

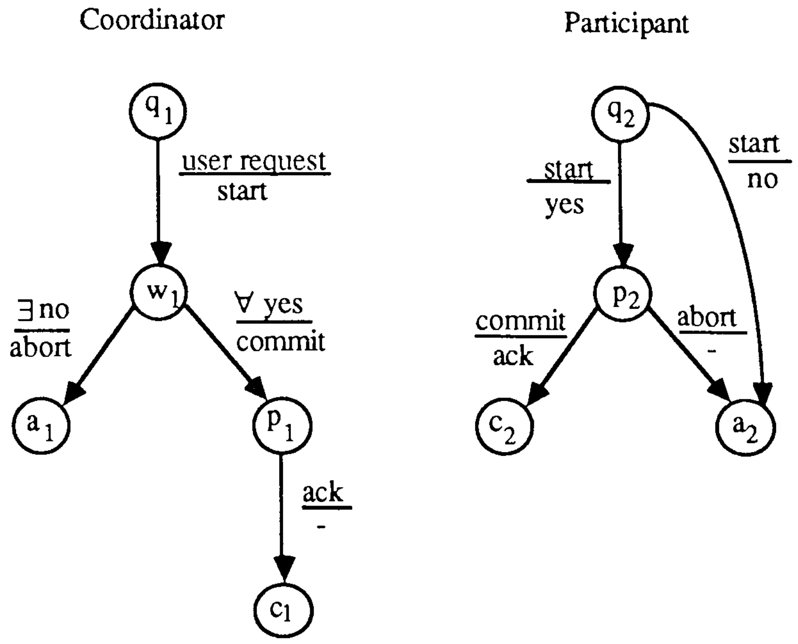


Fig.(3.2) FSM of extended 2 phase commit protocol

3.3.3. The 3 phase commit

Another well known protocol is the 3 phase commit described in section (3.2.2.1). This is the 2 phase commit extended with buffer states in the coordinator and participants as shown in figure(3.3). This protocol is similar to the extended 2 phase commit but will not be discussed further because although non-blocking it cannot be made independently recoverable.

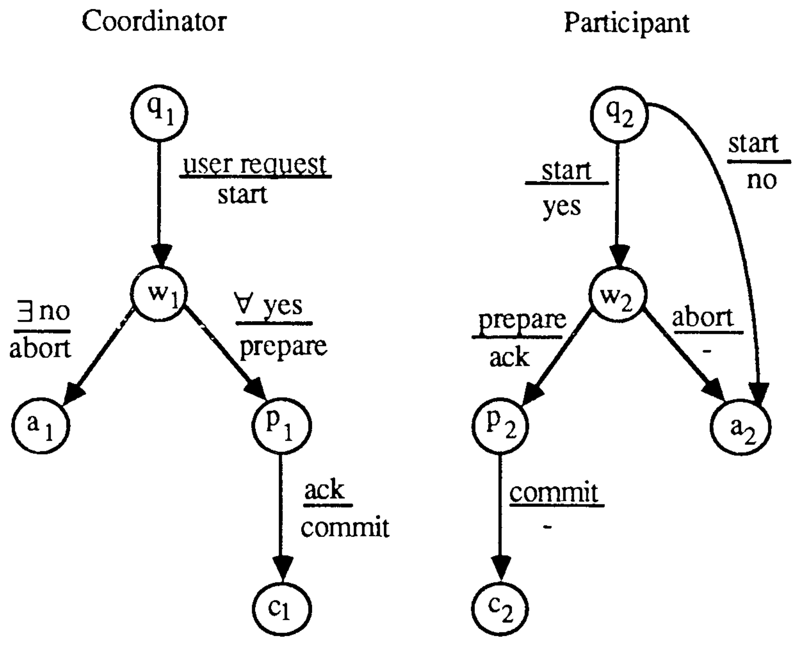


Fig.(3.3) FSM of 3 phase commit protocol

3.4. Analysis of the FSM models

As can be seen from the model of the 2 phase commit protocol, figure (3.1) each site progresses through a number of states until a final state is reached. The state occupied at any time by a single site is known as a local state. Collectively all the local states and any outstanding messages are known as the global state of the system. A global state is final if all the local states are final states, it is consistent if all the local states are the same. A global state changes every time a local state changes therefore for each local transition there is a global transition.

The global states achieved from a sequence of global transitions can be represented as a global state reachability tree. The states of the sites are shown above the line in the circles and any outstanding messages are shown below the line. The state trees for the 2 phase commit protocol and extended 2 phase commit protocol are shown in figure (3.4) and figure (3.5) respectively.

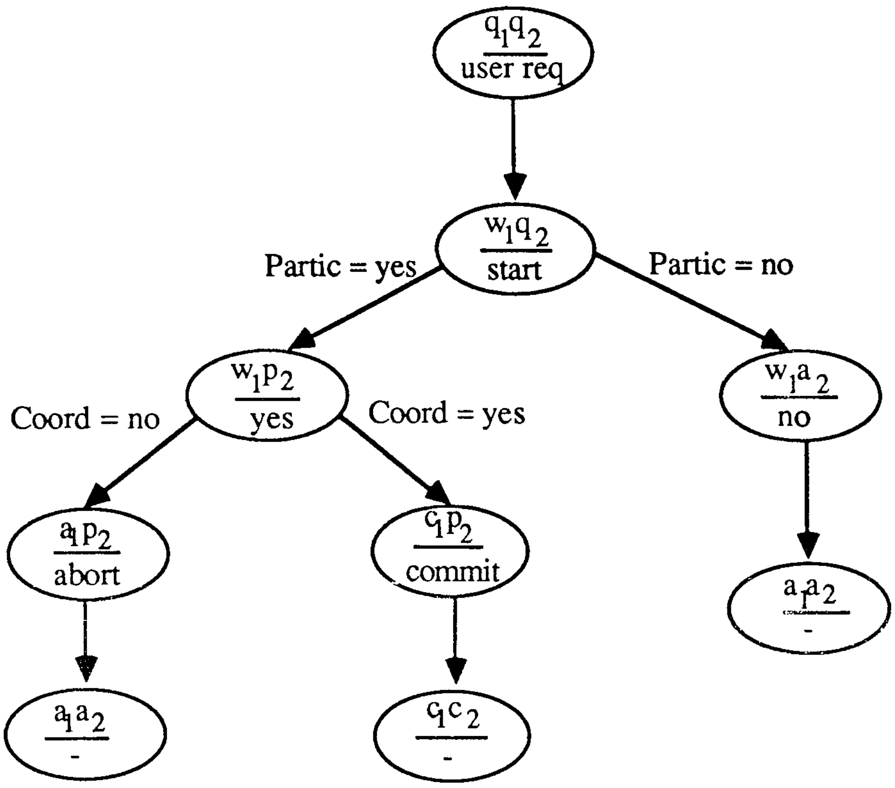


Fig.(3.4) Reachability tree for 2 phase commit protocol

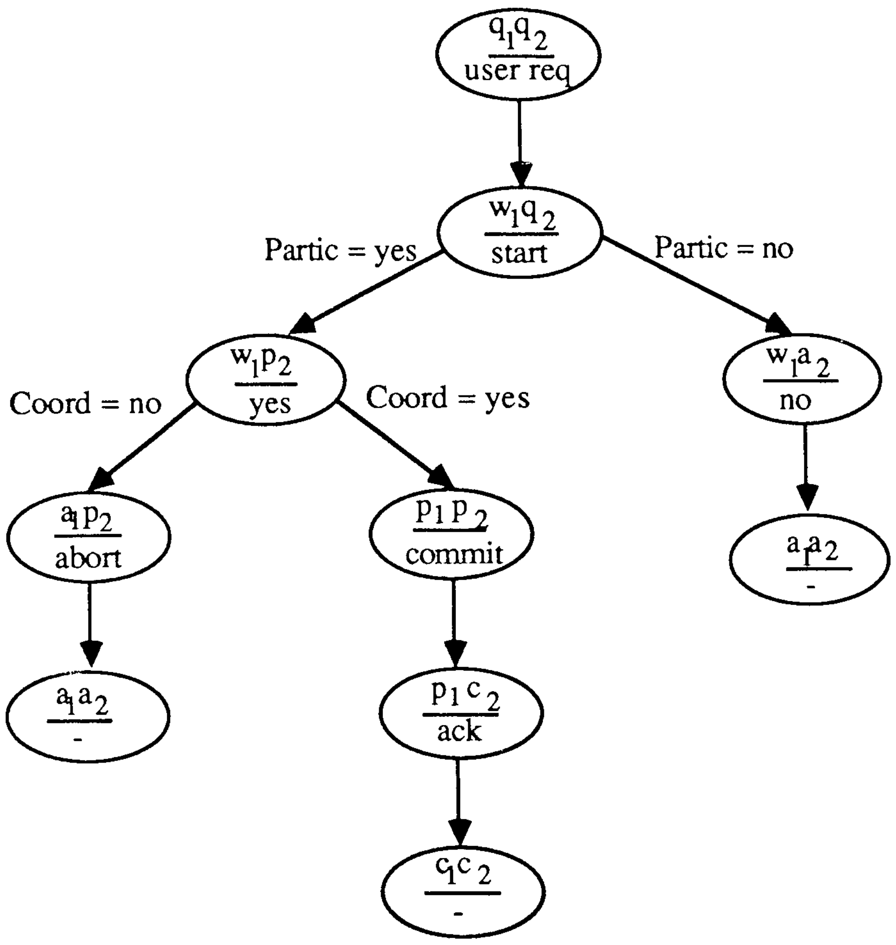


Fig.(3.5) Reachability tree for extended 2 phase commit protocol

Inspection of the reachability trees show that all the final global states comprise either all commits (ie c_1c_2) or all aborts (ie a_1a_2). Thus without any failures both of the protocols always terminate with a consistent global state. To analyse the state trees two notions are introduced, the concurrency set and the sender set. Skeen uses the idea of concurrency sets to determine if a protocol can be made independently recoverable. Sender sets are used so that the protocols can be made resilient to various failures by including timeouts.

A concurrency set for state S_i, $C(S_i)$ is defined as the set of all states that another site can occupy while site i is in state S_i. For example the concurrency set for w_1 in figure (3.4) is $\{q_2,a_2,p_2\}$. The concurrency sets for figure (3.4) and figure (3.5) are shown below :-

2 phase commit :-	Extended 2 phase commit :-
$C(q_1) = \{q_2\}$	$C(q_1) = \{q_2\}$
$C(w_1) = \{q_2, p_2, a_2\}$	$C(w_1) = \{q_2, p_2, a_2\}$
$C(a_1) = \{p_2, a_2\}$	$C(a_1) = \{p_2, a_2\}$
$C(c_1) = \{p_2, c_2\}$	$C(p_1) = \{c_2, p_2\}$
$C(q_2) = \{q_1, w_1\}$	$C(c_1) = \{c_2\}$
$C(p_2) = \{w_1, a_1, c_1\}$	$C(q_2) = \{q_1, w_1\}$
$C(a_2) = \{w_1, a_1\}$	$C(p_2) = \{w_1, p_1\}$
$C(c_2) = \{c_1\}$	$C(a_2) = \{w_1, a_1\}$
	$C(c_2) = \{c_1, p_1\}$

A sender set for state S_i, $S(S_i)$ is defined as the states which send messages to the site i when it is in state S_i, these sets are used for timeout placement. For example the state p_2 in the participant of figure (3.1) receives messages from the coordinator in state w_1, thus $S(p_2) = \{w_1\}$.

The sender sets for figure (3.4) and figure (3.5) are shown below :-

2 Phase commit :-	Extended 2 phase commit :-
$S(q_1) = \{\}$	$S(q_1) = \{\}$
$S(w_1) = \{q_2\}$	$S(w_1) = \{q_2\}$
$S(a_1) = \{\}$	$S(a_1) = \{\}$
$S(c_1) = \{\}$	$S(p_1) = \{p_2\}$
$S(q_2) = \{q_1\}$	$S(c_1) = \{\}$
$S(p_2) = \{w_1\}$	$S(q_2) = \{q_1\}$
$S(a_2) = \{\}$	$S(p_2) = \{w_1\}$
$S(c_2) = \{\}$	$S(a_2) = \{\}$
	$S(c_2) = \{\}$

To determine informally whether a protocol can be made independently recoverable the concurrency sets must be generated and analysed. If a local state contains both an abort and a commit state in its concurrency set then it cannot be made to recover independently. This is because it is impossible to determine the other sites actions without extra communication. The 2 phase commit protocol cannot recover independently because the concurrency set for state p_2 contains the states a_1 and c_1. Examining the concurrency sets for figure (3.5) shows that the extended 2 phase commit protocol is independently recoverable.

To make the extended 2 phase commit protocol resilient to failures, failure and timeout transition have to be included in the model. Ideally the failure transitions should model both site and communication failure. However, all previous work on FSM models of commit protocols assume that the communication mechanism is an asynchronous network in which messages can be lost. Theorem 1 (section 3.2.3), states that in such a system there is no protocol which is resilient to network partitioning following a communication failure. Therefore only site failures will be considered here. A site failure transition is assumed to be the action taken by a filed site when it recovers. This is usually by way of a recovery procedure which examines the log record and puts the site into a new state.

A timeout transition is the action taken by the other sites when they detect that a site has failed (eg a message has not been received). If the failure and timeout transitions are assigned using the following rules [Skeen 83] the protocol will always terminate consistently.

Skeen's rules

Rule 1: For each intermediate state s_i, if $C(s_i)$ contains a commit then assign a failure transition from s_i to a commit state else assign a failure transition from s_i to an abort state.

Rule 2 : For each intermediate state s_i, if t_j (the state of another site) is in $S(s_i)$ and t_j has a failure transition to a commit (abort) state then assign a timeout transition from s_i to commit (abort).

Applying these rules to the extended 2 phase commit protocol produces the FSM of figure (3.6) which can be analysed using the state tree to show that it always terminates consistently. The state tree is not shown here but the analysis is the same as for the Petri net model using asynchronous communication which is shown in chapter 4.

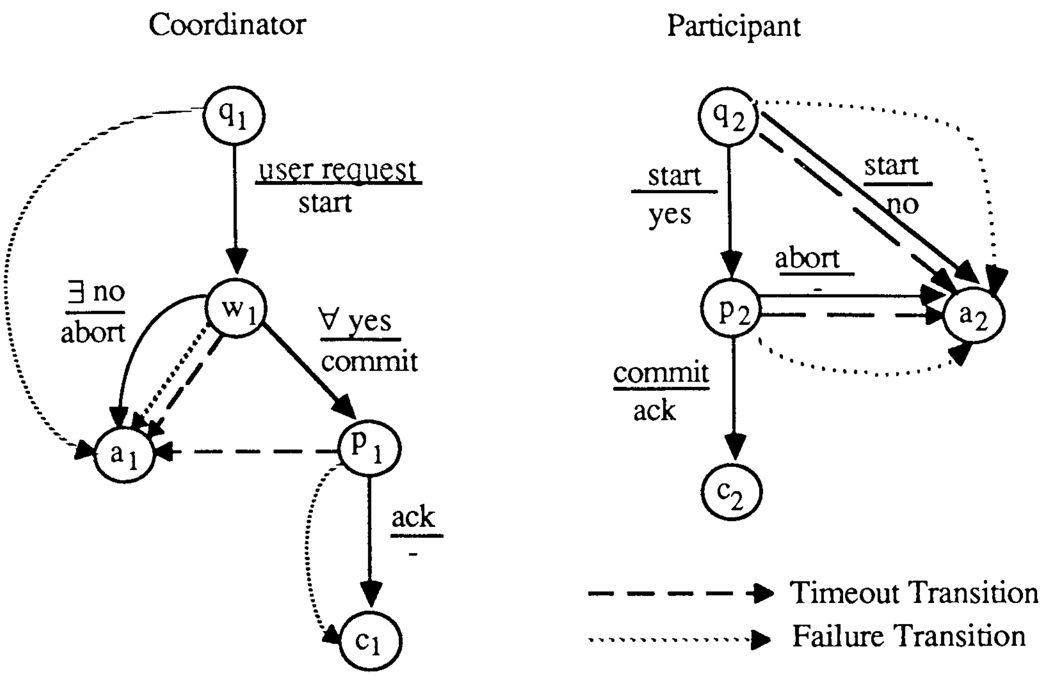


Fig.(3.6) FSM for extended 2 phase commit protocol resilient to site failures

3.5. Problems with modelling commit protocols using FSM

The FSM models of the previous commit protocols provide good state representation. However, they suffer from two problems which restrict their applications. Firstly they do not represent timing information and do not support the timing analysis required in real-time systems. Secondly they do not model the nature of the communications network which is central to an understanding of the behaviour of the protocols under failure conditions.

To demonstrate the limitations of the FSM communications model consider figure (3.6) and assume that the coordinator fails whilst in global state $\{p_1,p_2\}$ after the commit message has been received but before the ack is sent successfully. Upon recovering the coordinator enters state c_1, whilst the participant attempts to send the acknowledgement. Eventually the network will send a timeout command to the participant which will then occupy state c_2. The acknowledgement must now be absorbed by the network and forgotten.

This shows that the resiliency of the protocol is dependent on the network providing special features. Without a timeout the site would deadlock and attempt to send a message indefinitely. In a real-time embedded system this communication facility is not usually used, instead point-to-point communications are used. In such cases the onus for timeouts and dealing with unaccepted messages is placed within the protocols themselves. Such features should then be included within any model of the protocol.

Another problem with the FSM model is that state transitions are assumed to be atomic. This means that for the multi-site case if a coordinator sends messages to all the participants then if one of them received theirs all the participants received their message. This assumption is used because the communication system is not modelled and extra notation would be needed to remove it.

The above assumption was relaxed by Yuan [Yuan 89] who describes a multi-site protocol resilient to a single coordinator failing before sending all its messages. In this protocol when a site times out it sends messages to the other sites telling them what action has been taken, this has the disadvantages of more messages and that a fully connected network is required.

The rest of this thesis uses Petri net models which resolve the above disadvantages by allowing the communication system to be modelled as an integral part of the protocol. It also allows asynchronous and synchronous communications to be modelled and compared in such a situation. Another advantage of the proposed method is that timing information can be included in the model by allowing transitions to take a finite time. This allows intermediate timeout values to be estimated from an overall deadline for the decision. Also the Petri net models using synchronous communication map almost directly into the programming language Occam. This is an advantage because it means that there is no need to map the design into another model before implementation.

Chapter 4
The Design of robust atomic commit protocols

4.1. Introduction

The previous chapter has discussed previous work in the modelling of commit protocols. The problems with these techniques, in particular the addition of time and the omission of a network model have been high-lighted. One of the aims of this thesis is to apply commit protocols to real-time systems because they often need to coordinate tasks and reach agreement within predefined deadlines.

Any model used to specify the protocols must therefore be capable of including time, modelling the communication system whether synchronous or asynchronous, modelling site and link failures and also be easily transformed into an implementation. This chapter describes the formal model chosen, Time Petri nets, and shows how they can be used to design robust commit protocols which are augmented with timeouts so as to provide a decision within a deadline.

A Time Petri net is based upon the general Petri net with time included in the model. Petri nets are an alternative model to FSM which can also represent system state. They are most useful for modelling concurrent systems because parallelism and synchronization are easily modelled and understood. Petri nets can also be used to illustrate the dynamic properties of a system by execution of the net using a set of pre-defined rules, this results in a tree of all possible system states which is useful for analysis. Petri nets are also easily analysed for properties which are important to critical systems such as deadlock and state consistency. Unlike a FSM a Petri net is capable of modelling interactions between concurrent processes directly. This allows a model of inter-process communication that does not rely on assumptions about the transmission media. Thus they can model both synchronous and asynchronous communication very succinctly and accurately.

Since the protocols are to be applied to real-time systems, time must be included in the model, this is provided by extending Petri nets with time. Methods of extending Petri nets with time are described in section (4.2.1), this introduces the two basic models of Time Petri nets [Merlin 76] and Timed Petri nets [Ramchandani 74] and explains the difference. The resulting Time Petri net model can be analysed with respect to time as well as the state of a system and constraints made on the system so that consistent decisions are always made by a deadline. This represents a significant advance on existing approaches. For example commit protocols have been analysed using Finite State Machines (FSM) by Skeen and Stonebraker [Skeen 83] but deadlines were not included. The extension of commit protocols with timing constraints was first suggested by Holding, Hill and

Carpenter [Holding 88] for use in real-time systems. Similar recent work independently carried out by Davidson et al [Davidson 89] included deadlines in atomic commit protocols but they do not allow for failures or include a formal model.

Petri nets allow the communications between sites to be explicitly modelled thus showing any differences in using asynchronous or synchronous primitives. Section (4.3) contains the Time Petri net models for the commit protocols described in chapter 3. Both asynchronous and synchronous communications are modelled so that comparisons can be made. The protocol using asynchronous communications is shown to be isomorphic with the original FSM model. The Time Petri net models are then augmented with site and link failures and their resiliency is investigated. Novel work in this section includes the design of commit protocols using synchronous point-to-point communication primitives and the addition of time to provide decisions within deadlines. From the protocol using synchronous communications the previous rules (used for the FSM models) for placing timeout and failure transitions are shown to fail. A new method of placing them by examination of the state reachability tree is provided in section (4.4). Section (4.4) also includes the analysis of the nets where properties such as consistency and freedom from deadlock are shown. Properties involving time such as timer evaluation are also included.

The protocols designed in this chapter using synchronous communications have been implemented in Occam and executed on a network of Transputers, these implementations are described in chapter 5. The resiliency of the protocols to site and communication failures is shown to follow the analysis of the Petri nets. The full listings for the 2 phase commit (blocking) and extended 2 phase commit (non-blocking) protocols with timeouts derived for link failures are provided in Appendix B.

The Time Petri net models developed also allow site and link failures to be modelled in a uniform manner. They are included in the specification which is used to determine the placement of recovery mechanisms. Failure transitions are added to simulate faults, this was first used by Merlin [Merlin 76] for communication protocols and later used by Leveson [Leveson 87] for safety critical systems. They are now applied here to commit protocols under failure conditions. Modelling the communication system allows a lower level of atomicity for message transfer than previously assumed [Skeen 83] and permits non-atomic communication. The Petri net model is novel in the respect that timing and failures are added and also the communication system is modelled. Timing is included to investigate the timeliness of decisions when failures occur.

It is shown how protocols can be designed to be resilient to either site failures or link failures but not both. Protocols resilient to link failures are very important for control systems because when a link failure occurs each site (processor) can still control its application autonomously and possibly provide a safety feature. When a site fails the control of its actuator is lost and so safety features must be provided by either the other operational sites or backup sites. Thus deadlines can be provided for link failures but due to unpredictable recovery time deadlines cannot be met with site failures. Concluding remarks are provided in section (4.5).

4.2. Petri nets and time

A Petri net [Peterson 81, Murata 89] is a formal model used to study systems, in particular concurrent systems can be modelled extremely well. They are based upon an extension of set theory known as bag theory which allows elements to be repeated any number of times. Two representations exist [Peterson 81], the Petri net structure and the Petri net graph, when referring to 'Petri nets' the later is implied.

A Petri net structure is a 4-tuple, (P,T,I,O) where P is a set of places, T is a set of transitions and I and O are input and output functions. The input function maps a transition into a number of input places whilst the output function maps a transition into a number of output places. Using bag theory instead of set theory allows places to be multiple input or multiple outputs. This representation of a Petri net is convenient for manipulation by computer but for most purposes a Petri net graph is more useful.

A Petri net structure can be transformed directly into a Petri net graph [Peterson 81], which is the usual representation. A Petri net graph is a bipartite directed graph which represents places (circles) and transitions (bars) as nodes. The nodes are connected by directed arcs which represent the input and output functions.

The state of a system can be modelled using a Petri net graph by allowing tokens which reside within appropriate places to represent the instantiation of a particular state. Any number of tokens are allowed to reside within a place. The set of places and the number of tokens in each place are known as the marking of the net. An example of a marked Petri net, in which the tokens are denoted by black dots is shown in figure (4.1).

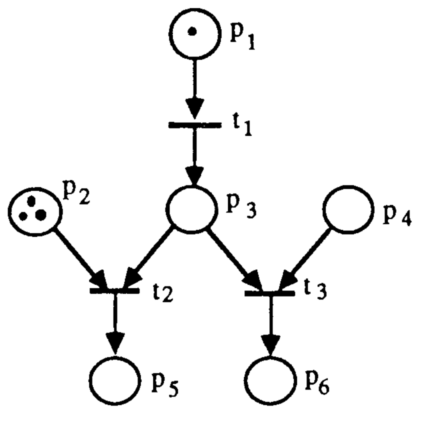


Fig.(4.1) Example of a marked Petri net

A general Petri net allows any number of tokens to reside in any place at any time (as in place p_2 of figure (4.1)). However, the Petri nets used in this thesis are a special class of Petri nets known as safe Petri nets [Murata 89] because the number of tokens in a place may not exceed one. This simplifies the marking of the net into a set of places. The initial marking of figure (4.1) is $(p_1,p_2,p_3,p_4,p_5,p_6) = (1,3,0,0,0,0)$ whereas if the Petri net was safe and p_2 only had one token the marking would be (1,1,0,0,0,0) and denoted as (p_1,p_2).

A marked Petri net can be executed following a set of rules. Basically a transition fires if all its input places contain tokens. When the transition fires, one token is removed from each input place and a token is added to all the output places. For example in figure (4.1) t_1 can fire which removes the token in p_1 and places one in p_3 producing a new marking of (0,3,1,0,0,0). The firing of transitions continues until no more are enabled, this is known as the execution of the Petri net. In a safe Petri net which models concurrent systems there will be one token per parallel process. During the execution many transitions may be enabled at the same time thus allowing nondeterminism. This results in a number of possible executions for the same Petri net.

Two techniques exist for analysing Petri nets, using matrix equations [Murata 89] and using the state reachability tree [Karp 69]. The matrix representation is very applicable for computer manipulation of a Petri net but is not as explicit in its state representation as the reachability tree. Although the reachability tree cannot in general solve all reachability or liveness problems, in the restricted class of Petri nets that are used in this thesis it can explicitly show deadlock and state consistency. This shall therefore be the technique used for analysing the protocols.

The reachability tree is generated by executing each enabled transition and recording the new markings produced. Where more than one transition is enabled extra branches are added to the tree, each branch representing the firing of a different transition. A complete reachability tree shows all the possible states that a system may exhibit during its execution. As an example consider the reachability tree shown in figure (4.2), this shows all the states that the Petri net of figure (4.1) may posses.

$$\begin{tikzcd}
(1,3,0,0,0,0) \arrow[d, "t_1"] \\
(0,3,1,0,0,0) \arrow[d, "t_2"] \\
(0,2,0,0,1,0)
\end{tikzcd}$$

Fig.(4.2) Reachability tree for fig.(4.1)

When modelling software with Petri nets [Mekly 80] it is common for places to represent conditions and transitions to represent events (e.g. assignment). The Petri net reachability tree then expresses causality in that certain conditions must prevail before an event can occur. As an example, in figure (4.1) the conditions p_2 and p_3 must hold before the event t_2 can occur. The reachability tree features substantially in this thesis for analysis of the commit protocols developed.

4.2.1. Time Petri nets

There has been much work into extending Petri nets with time resulting in a number of different ways of including and expressing timing delays. The two most used methods are those of Timed Petri nets [Razouk 84] and Time Petri nets [Merlin 76]

Timed Petri nets were originally defined by Ramchandini [Ramachandini 74] as having a fixed delay with each transition. This delay was later applied to places [Sifakis 77, Coolahan 83] so that the original definition of instantaneous transitions was obeyed. The timing was decomposed for each transition, t_i into an enabling time, $E(t_i)$ and a firing time $F(t_i)$ by Razouk [Razouk 84] so that the dynamic behaviour of the Petri net could be described better and new rules for the firing of a transition which involved time devised. A transition is ready to be fired at time T if it has been enabled continuously for the period $[T - E(t_i), T]$. At the point T the transition fires absorbing tokens from its input places. Unlike normal Petri nets tokens do not appear in the output places until after the time $T + F(t_i)$, during the period $[T, T + F(t_i)]$ the transition is said to be firing.

Performance analysis is derived by using the timed reachability tree [Zuberek 80, Razouk 85a], these can become very complicated but are not required for deriving timing constraints. These Timed Petri nets have been used to design deterministic timeouts for use in a communication system [Suzuki 90].

Time Petri nets were developed by Merlin and Farber [Merlin 76] and involve associating a minimum, Tmin (t_i) and a maximum, Tmax (t_i) firing time to each transition. Transitions are allowed to fire some time in the interval [Tmin (t_i), Tmax (t_i)] after it first becomes enabled. If the time Tmax (t_i) is reached before firing the transition is guaranteed to fire. This model is more flexible and also models timeouts more accurately because a timeout cannot be timed exactly. It is more flexible because the tokens are not removed from the places until the transition has fired, this simplifies reachability analysis. The original definition of a transition firing being instantaneous is also obeyed.

An analysis technique similar to the reachability analysis of standard Petri nets has been developed by Berthomieu and Menasche [Berthomieu 83]. This allows a finite representation for the possibly infinite number of next states involving time. The technique is not used to determine timing constraints. Time Petri nets have been used by Leveson and

Stolzy [Leveson 87] to derive timing constraints so as to avoid high risk states. The models in this thesis are based upon the Time Petri net model of Merlin and Farber because of its flexibility and simplicity.

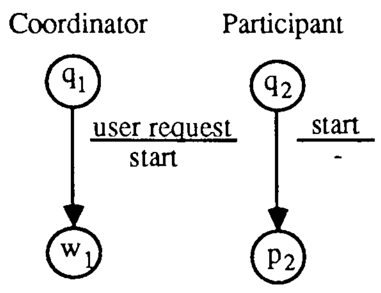
4.3. Time Petri net models of commit protocols

This section shows how the commit protocols described in chapter 3 can be modelled using Time Petri nets thus removing the disadvantages of the FSM model. Firstly it shows how the communication system can be modelled, the protocols are then modelled incorporating asynchronous and synchronous communications. The protocols chosen are the 2 phase commit and the extended 2 phase commit. The 2 phase commit protocol is chosen because it is the basic commit protocol and is useful for comparing the FSM and Petri net models. The extended 2 phase commit protocol is modelled because it can be made independently recoverable which is a very important feature when considering real-time applications. The extended 2 phase commit is also an optimal non-blocking protocol [Chin 87]. Finally, it is shown how site failures and communication failures can be modelled using Petri nets.

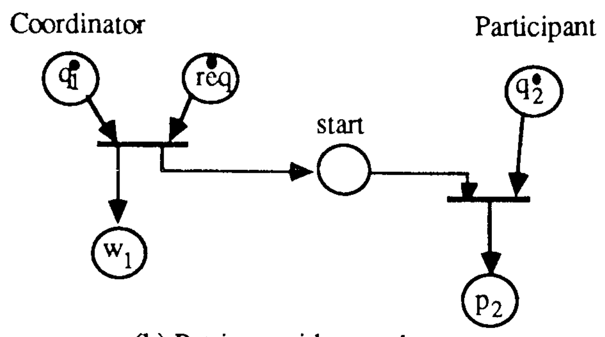
The protocols are modelled with both asynchronous and synchronous communications and analysed using a reachability tree to show that they are still correct when there are no failures. The analysis of the protocols when failures occur and time constraints are applied is left until section (4.4).

4.3.1. Modelling the communication system

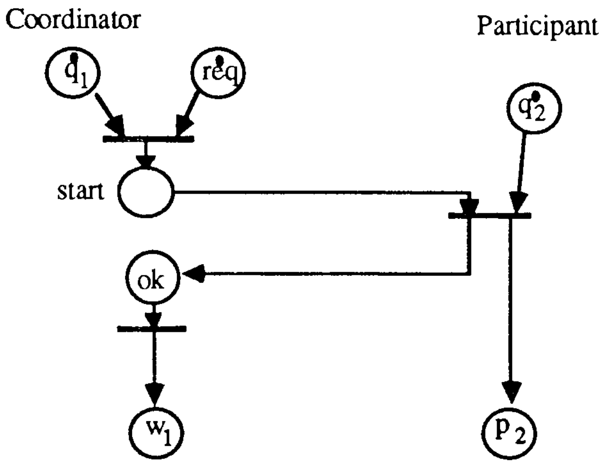
A FSM models the state of a system but omits detail about the communications. Petri nets can model as much state information as an FSM and also model the communications. Both asynchronous and synchronous communications can be modelled thus allowing the two to be compared. Figure (4.3) shows how a two site FSM that receives a message and sends another can be transformed into Petri nets, figure (4.3b) shows the asynchronous communications whilst (4.3c) shows the synchronous version.



(a) Finite state machine



(b) Petri net with asynchronous communications



(c) Petri net with synchronous communications

Fig.(4.3) Modelling communications

This shows that there are two different actions depending upon the type of communications used. In the asynchronous communication model, a message is sent and the sender changes state concurrently. However, in the synchronous model a state transition of the sender only occurs when it knows that the message sent has been received.

Tokens in the places start and ok in figure (4.3c) represent that the sender is ready to send the start message and that the message has been received successfully. This is the ideal model of synchronous communications and in reality cannot be implemented identically.

Synchronous communications can be implemented using a handshaking technique, such as used by Inmos Transputer links [May 85]. This involves the sending of two messages, the actual data to be transmitted and an acknowledgement signal. The acknowledgement is returned by the receiver after the original message is received. Once the acknowledgement is received, the sender knows that the original message was received successfully and that another message can be sent.

As an example, the coordinator in figure (4.3c) will only reach state w_1 after the communication has taken place, i.e. a token appears in the ok place. This only happens after the 'start' message has been sent and acknowledged.

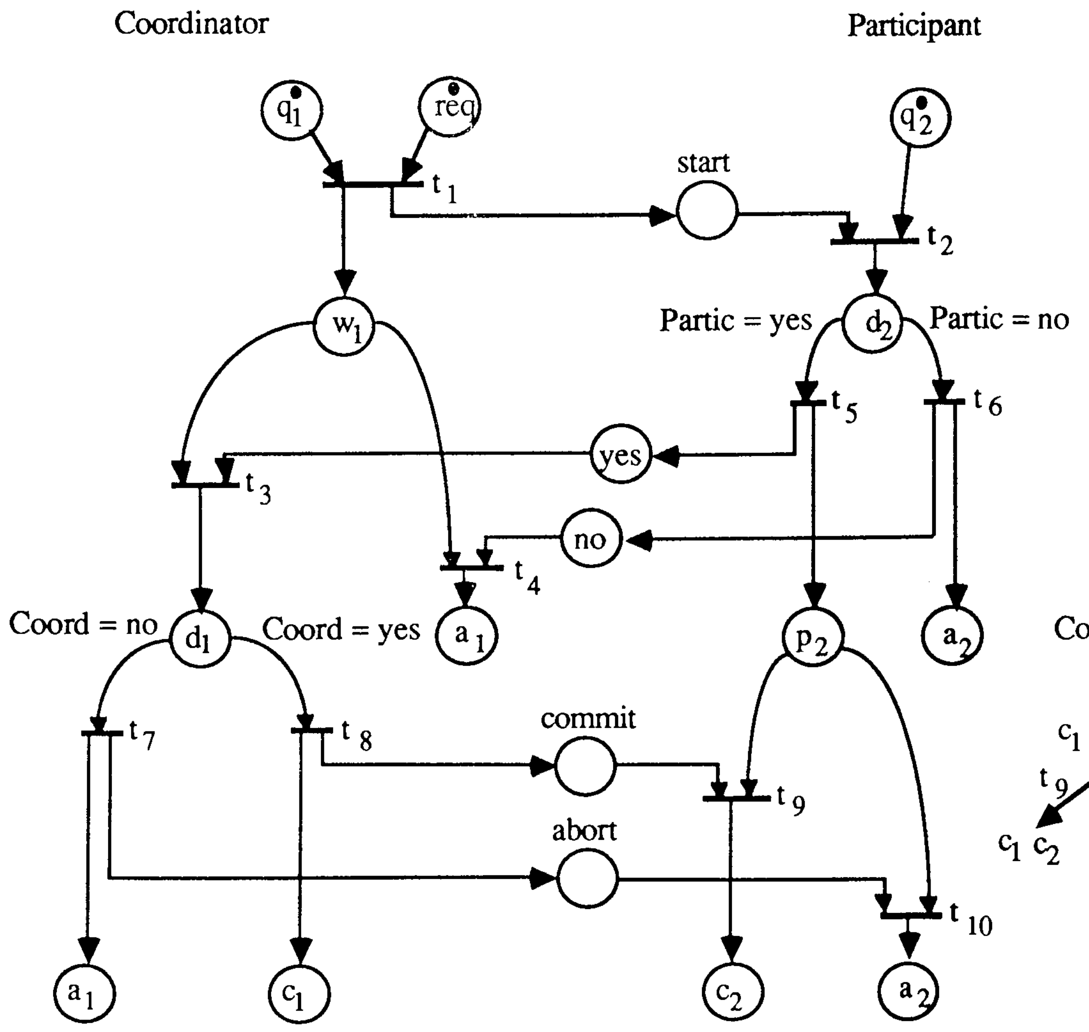
4.3.2. Two phase commit Petri net models

The 2 phase commit described in section (3.2.1.1) can be modelled using Petri nets showing state information and the communication system. The Petri net specification using asynchronous communications is shown in figure (4.4) complete with the associated reachability tree.

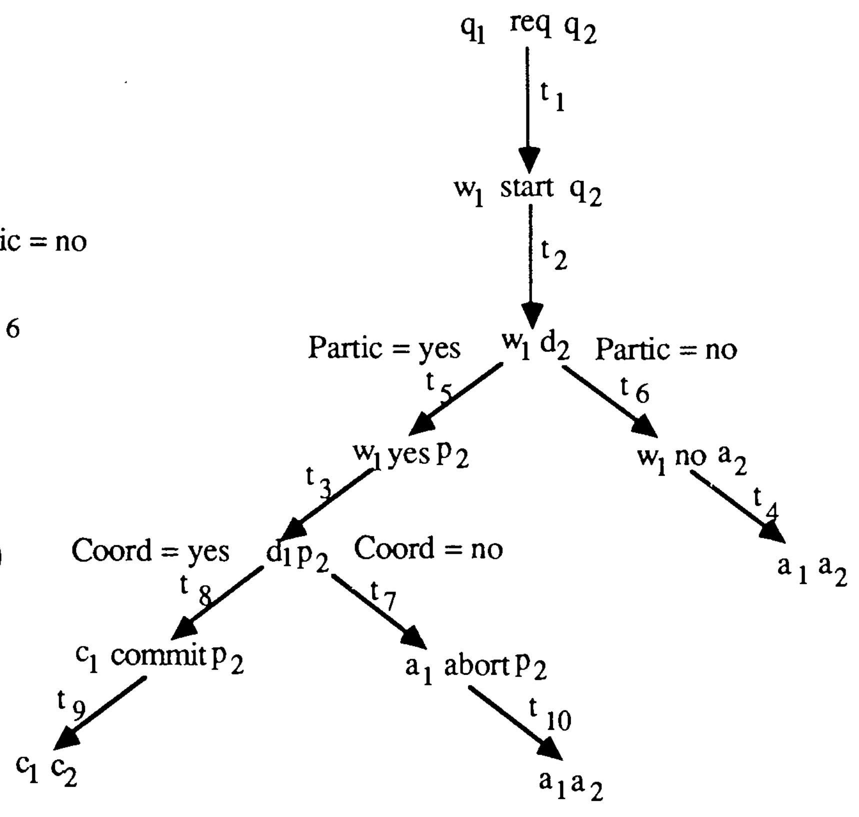
The place marked 'req' is an external request used to initiate the protocol and can be assumed to always happen. When a token is placed in a message place this means the message is being sent, it is equivalent to the outstanding message in the FSM reachability tree. One advantage of showing the messages explicitly at this level is that communication failures can be modelled.

The reachability tree shows that without failures a consistent final global state is reached. As for the FSM model, the concurrency set for each state can be calculated. In this case the concurrency set for the state p_2 is $\{w_1, a_1, c_1,$ yes, commit, abort$\}$ again showing that the protocol is blocking. The sender sets do not need to be used because they are obvious from the Petri net.

It is expected that the protocols are to be implemented using synchronous communications but figure (4.4) does not model this type of communications. Thus another model must be derived that explicitly shows synchronous communications. This can be developed from the Petri net of figure (4.4) by replacing each asynchronous communication with a synchronous communication, thus deriving the Petri net of figure (4.5).



(a) Petri net



(b) Reachability tree

Fig.(4.4) 2 phase commit - asynchronous communications

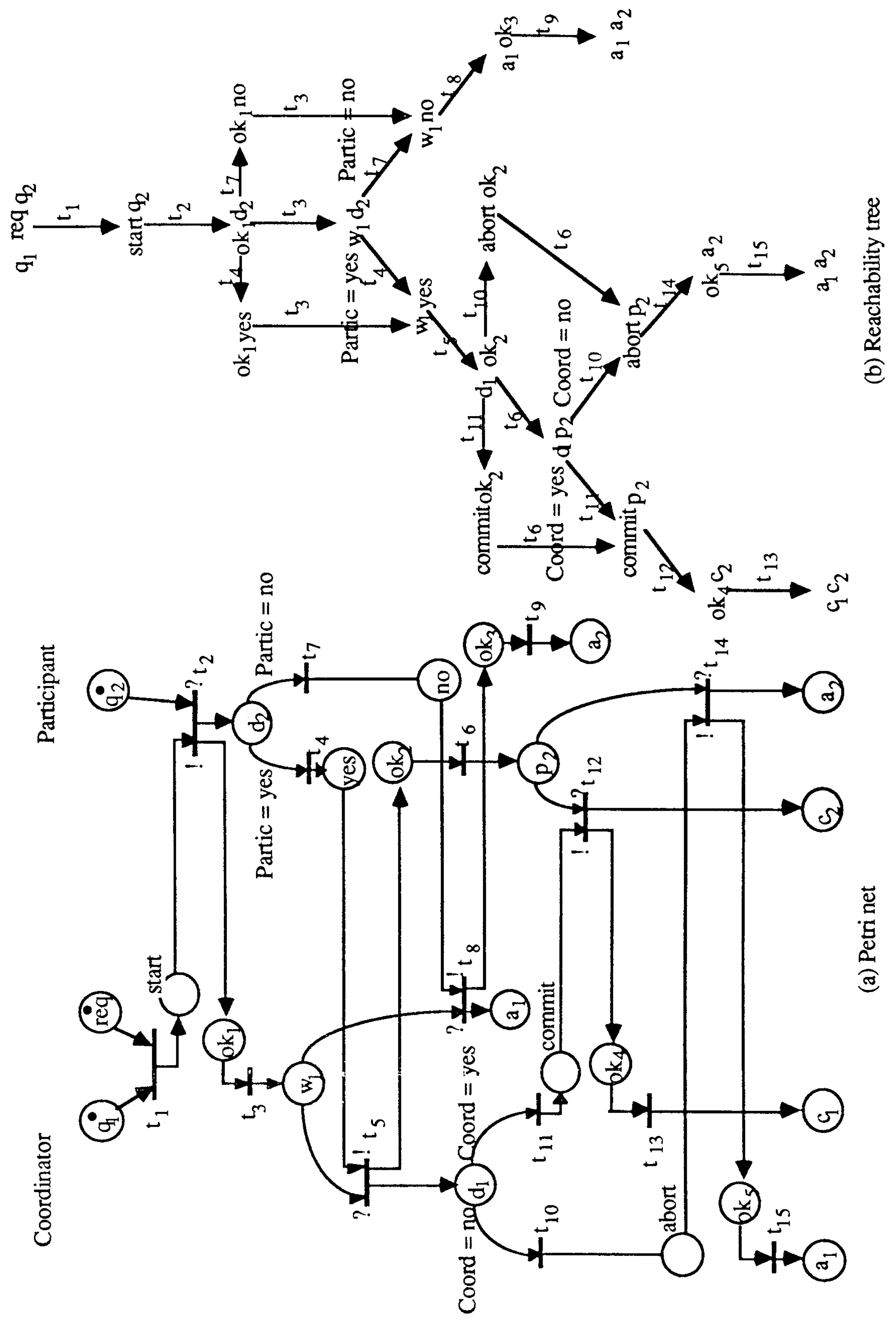


Fig.(4.5) 2 phase commit - synchronous communications

Each ? and ! pair represent a single synchronous communication and shows how the synchronous communication primitives embedded in the notions of CSP [Hoare 78] and Occam can be used in the modelling of the protocol. If the implementation of synchronous communications is by handshaking then each ? and ! pair will comprise of two messages. Although the protocol only involves five communications, ten actual messages would be used.

The reachability tree of figure (4.5b) can be reduced to simplify its analysis. The reduction is possible because it can be assumed that any implementation of synchronous communication acknowledges the received message before any other transition can occur. This preserves the idea of synchronous communications as closely as possible. The reduced reachability tree for this for figure (4.5a) is shown below in figure (4.5.c). All subsequent protocols using synchronous communications follow this assumption and are shown only with the reduced reachability trees.

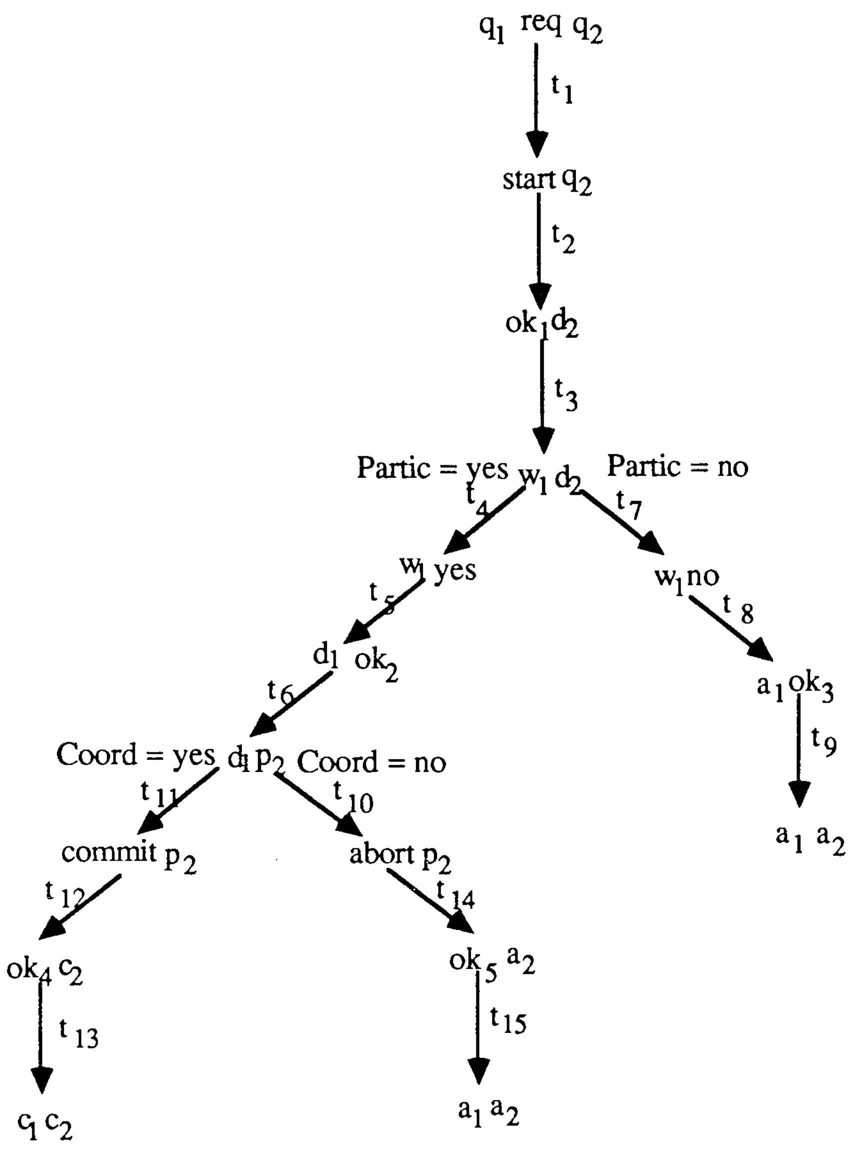


Fig.(4.5c) Reduced reachability tree for Fig.(4.5a)

Using the reduced reachability tree does not affect the analysis of the protocols because the omitted states should never be reached in an implementation. From this reduced reachability tree the concurrency sets can be obtained, none of which contain both commit and abort states, thus this protocol should be non-blocking. On further investigation this property is only gained because each part of the synchronous communication is shown separately. For example the abort/commit messages are both shown separately. In practice, where such a protocol is implemented, for example in Occam on a network of Transputers, only one communication link would be used. This means that the places ok_4 and ok_5 are in fact the same meaning that the transmitted message has been received. Now $C(ok_4) = \{a_2\}$ and $C(ok_5) = \{c_2\}$ therefore since $ok_4 = ok_5$, then $C(ok_4) = \{a_2, c_2\}$ and the protocol is blocking as expected.

From the two Petri nets it has been shown that the blocking property of the 2 phase commit is independent of the type of communications used. Blocking protocols do not have a bounded recovery time and are not considered useful for real-time systems.

4.3.3. Extended 2 phase commit Petri net models

The 2 phase commit has been extended using an acknowledgement so that it is non-blocking to site failures [Skeen 83], the resulting protocol is known as the extended 2 phase commit. This section shows the Petri net models of this extended protocol to which later sections will add failures and timeouts. Figure (4.6) shows the protocol using asynchronous communications whilst the synchronous version is shown in figure (4.7). The reachability tree of figure (4.7b) is of the reduced form as described earlier for the synchronous 2 phase commit protocol.

Analysing the reachability trees show that without failures both versions are non-blocking. The effects of including site and link failures to both these is left until section (4.4). The synchronous version is non-blocking even taking into account the problem explained in the last section.

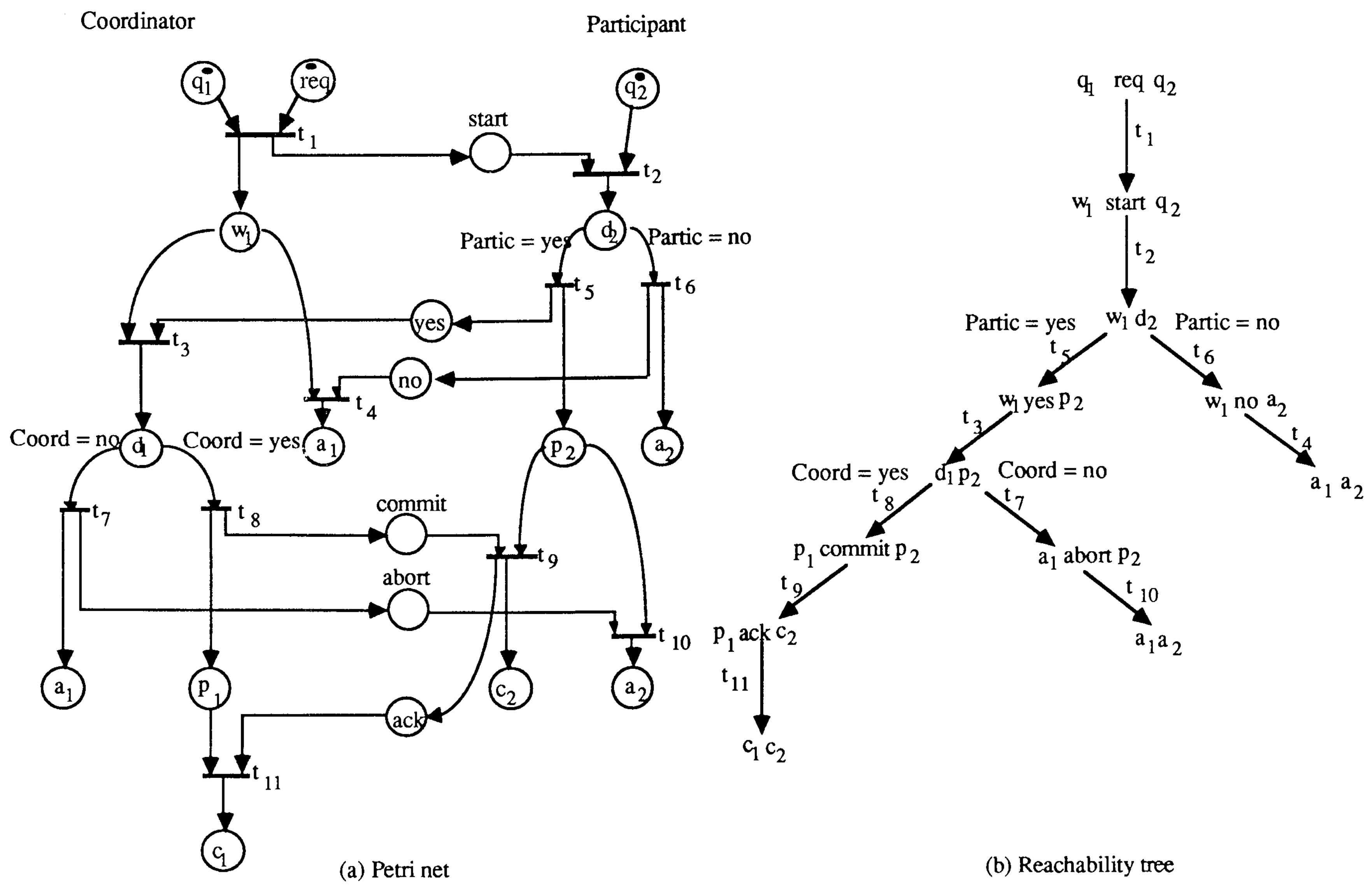
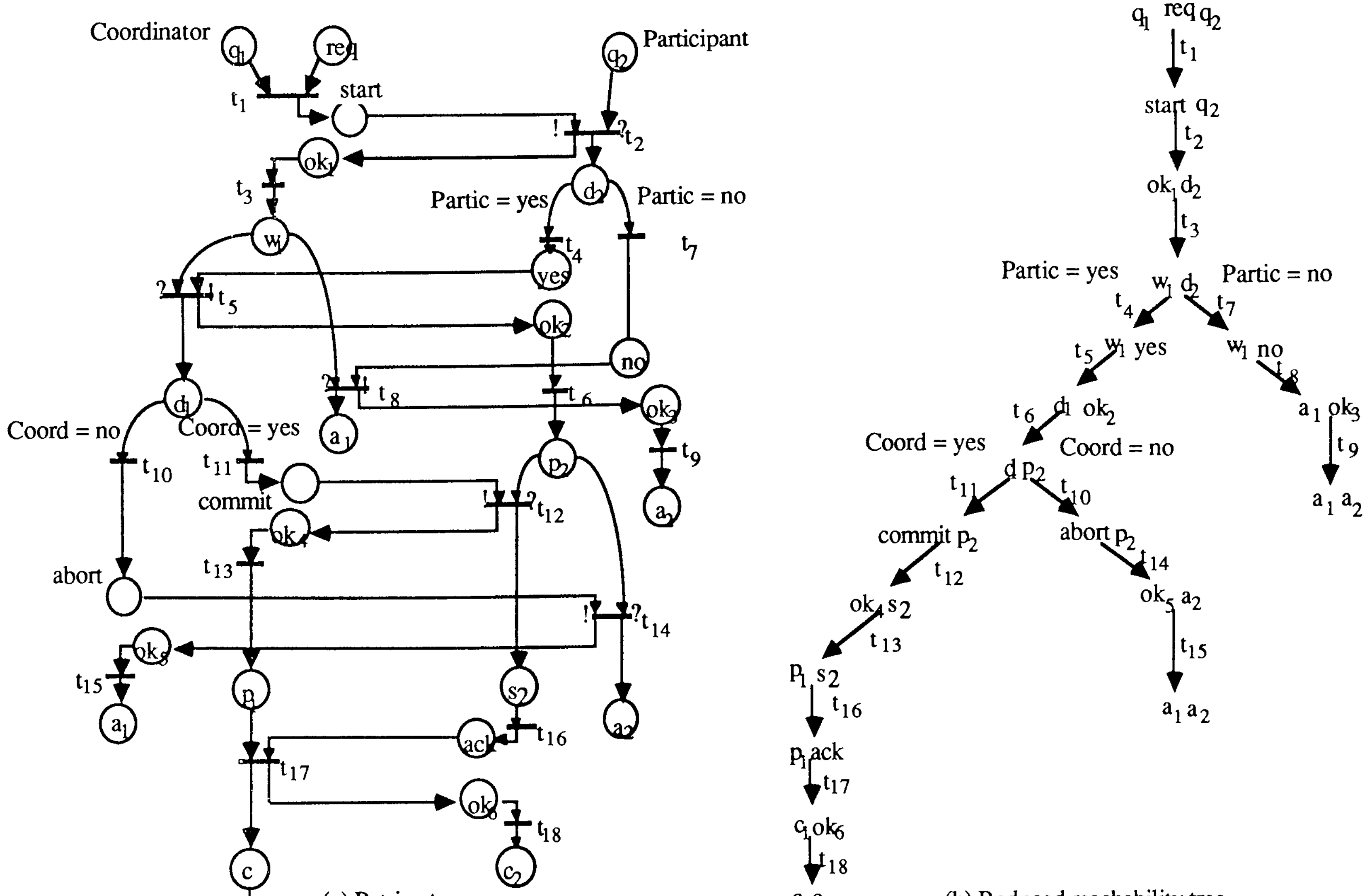


Fig.(4.6) Extended 2 phase commit - asynchronous communications



(a) Petri net

(b) Reduced reachability tree

Fig.(4.7) Extended 2 phase commit - synchronous communications

4.3.4. Possible Failures

The commit protocols previously discussed are all correct when there are no failures but in a real system the protocols also need to be correct when failures occur. Failures have been classified into failures of commission or omission [Mohan 83b].

Commission failures are where incorrect actions are performed by the failed item, these type of failures are very difficult to rectify and are not considered in this thesis. The resolution of commission errors is similar to solving the Byzantine generals problem [Lamport 82], this has been applied to commit protocols [Mohan 83b] the problems of which were outlined in section (3.2.1.2).

Omission failures are when a failed item does not do something that is expected, this is sometimes known as the fail-stop approach [Schneider 87] and is much simpler to deal with. Only omission failures are dealt with in this thesis, these can be further classified into site and communication failures.

A site failure is considered to be safe when no erroneous computation are executed, the site just stops processing. In this thesis each site is assumed to be running a protocol written in a high level language so the atomicity of failure is at this level, this means a high level statement is either fully executed or not at all. The lowest level of atomicity would be at the machine instruction level but this is not considered here.

To simulate a site failure a recovery procedure is used. A call to this procedure is inserted where the failure is to occur and the remaining code is ignored. The recovery procedure is assumed to be executed immediately a failure is detected but a delay can be included to simulate repair time. In reality recovery involves reading a local log file and returning the site to a state consistent with the other sites. Depending upon the protocol used, recovery could involve extra communication between sites. When no extra communication is needed this is known as independent recovery and has been discussed earlier.

A site failure can be modelled using Petri nets. Consider for example the failure of the coordinator as modelled in figure (4.8).

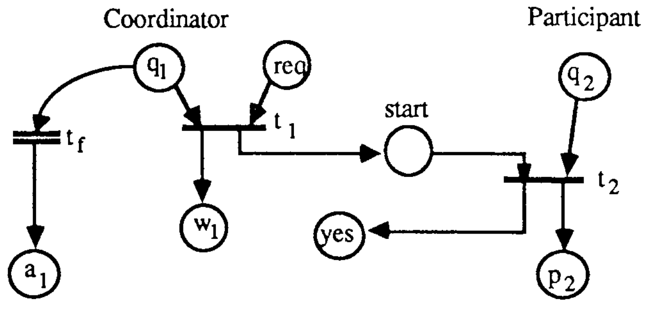


Fig.(4.8) Example site failure

Here a failure is assumed to occur at the coordinator site whilst in state q_1 if the failure transition t_f fires before t_1. It is assumed that a recovery protocol is executed and the coordinator aborted. The participant on the other hand is in state q_2, it cannot continue and is deadlocked. This situation is undesirable and the use of timeouts is described in section (4.4) to avoid it.

There are a number of possible communication failures depending upon the network used. In general message loss, out of order messages and link failure can occur. In an asynchronous system all of these are probable but with synchronous communications, such as those used in a network of Transputers, the synchronous nature of the communications prevents message loss and out of order messages. Therefore the only type of communication failure considered in this thesis for the synchronous communication model is the inability for two sites to communicate. Since the protocols in this thesis assume that there is only one communication path between two sites total communication failure can be caused by a physical link failure.

Depending upon the environment in which a system is applied then link failure may be the most probable cause of a failure. In particular within a control environment if a number of processors are distributed, say on a robot arm then the inter-processor links may become trapped and severed. Protocols resilient to this kind of failure are therefore very important to prevent unsafe conditions in particular in safety critical systems. With synchronous communications a sender site knows if a receiver has received a message or not, therefore if a link failure occurs before the acknowledgement was returned then the sender knows its message was not received.

A network partition occurs when a link failure separates the network into two or more isolated groups of sites. The sites in each group can communicate between each other but no communications between groups can occur. A simple partition is where only two groups are isolated from each other. Multiple partitions are not considered because there does not exist any protocol resilient to them [Skeen 83].

It was pointed out by Skeen [Skeen 83] that a multi-site protocol resilient to a single partition is only possible if undeliverable messages are returned to the sender, this implies it should be possible to design protocols in languages containing synchronous communications such as Occam which are resilient to a single link failure. It should be noted here that no protocols exists for multiple partitions or where messages are lost.

In this thesis, synchronous protocols are implemented using the concurrent programming language Occam which is based on CSP. A link failure in the Occam simulations is provided by removing the actual link during execution. A link failure can be modelled in the Petri nets by a transition that never fires as denoted by the double bar in figure (4.9).

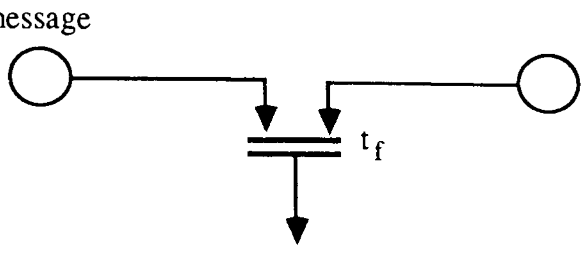


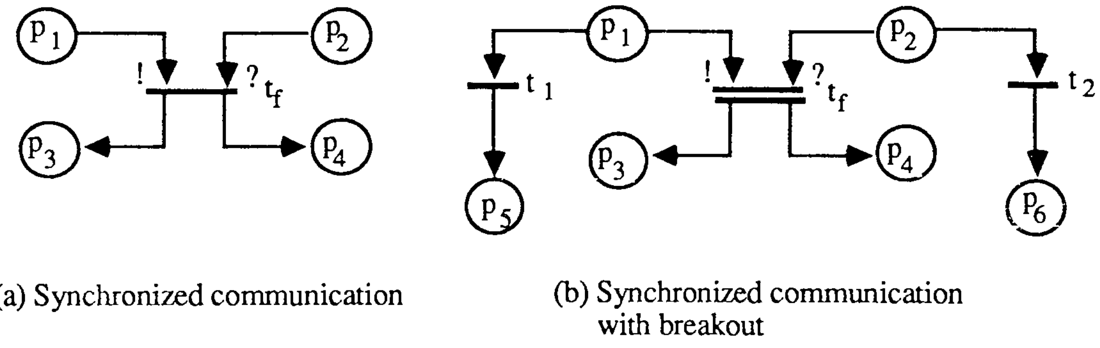
Fig.(4.9) Example of a link failure

With asynchronous communications message loss can occur, this can be modelled by a token being removed from a message place [Merlin 76], this is not considered appropriate for the models using synchronous communications.

4.3.5. Deadlock prevention

Deadlock in a database means a transaction is waiting to lock a data item which will never happen because other transactions may have conflicting locks on it. In this thesis deadlock is used to mean a process is waiting indefinitely because of some site or communication failure. For instance if a site fails before sending a message the other process will wait for the non-existent message. This is obviously an undesirable property when a real-time system and deadlines are considered. This situation can be avoided by the use of timeouts and is described here for the use with synchronous communications.

Consider a message being passed synchronously between two processes as shown in figure (4.10a). If the link fails then to prevent deadlock the tokens from P_1 and P_2 must be able to bypass t_f, this is achieved by having a timeout on both sides of the communication as shown in figure (4.10b).



(a) Synchronized communication

(b) Synchronized communication with breakout

Fig.(4.10) Communication timeouts

There are a number of problems with using timeouts to prevent deadlock, firstly false timeouts may occur and secondly there are a large number of timeouts whose values must be controlled. The values must be controlled so that one timeout does not force another false one and also so that an overall deadline is still met. False timeouts are due to not allowing enough time for the communication to occur. The worst case for a timeout value must always be chosen to allow enough time for process synchronization and the message propagation delay of the link. The timeout value must not be too large because undue delays are unacceptable and may upset the timing requirements of other parts of the system. The number of timeouts required for synchronous communications is quite large because every input and output must be bounded, also it must be ensured that there are no redundant timeouts and also that timeouts do not depend on others.

It has been suggested that time limits can be added to the transitions in a Petri net [Merlin 76, Leveson 87]. Each transition is given a minimum and a maximum firing time after the transition is first enabled. Thus to allow enough time for communication in figure (4.10b), the condition

$\mathrm{Min}\ (t_1) > \mathrm{Max}\ (t_f)$ and $\mathrm{Min}\ (t_2) > \mathrm{Max}\ (t_f)$

ensures that the communication has adequate time to execute if possible. Since communication is synchronized the first primitive to be ready (? or !) must wait for its corresponding partner, this may be quite a way back in its execution and so this time must be allowed when calculating timeout values such as t_f. That is :-

$\mathrm{Max}\ (t_f) >= \mathrm{MAX}[\text{message propagation delay}] + \mathrm{MAX}[\text{Process synchronization delay}]$

A similar expression can be derived for Min (t_f) using the corresponding minimums. Since these values are predictable accurate timeout values can be chosen and deadlines kept.

4.4. Failure and timing analysis of the models

The 2 phase commit has been shown to be a blocking protocol under certain failures. Since deadlines are being applied only the extended 2 phase commit will be analysed here. This is because it can independently recover and it is also an optimal non-blocking protocol. Firstly a two-site model is used and site failures allowed at either site. Secondly link failures are introduced. Link failures will be emphasized because if a protocol can independently recover from this kind of failure then processing can continue independently at each site thus providing any required safety features. It is shown that the Petri net model does not introduce any problems and so the asynchronous version of the extended 2 phase commit is analysed and it is shown that this provides the same results as Skeen's FSM model. Skeen's rules (see section (3.4)) for timeout placement are shown correct for the asynchronous model and investigated for the synchronous model.

The synchronous model is used to show how the communication structure affects timeout placement and that Skeen's rules are not applicable to synchronous protocols. A new set of rules are provided so that the synchronous protocols can be made resilient. The synchronous model includes redundant messages and this fact is used in chapter 5 for optimizing the protocols.

The analysis introduces site and link failures into the model so that the reachability tree can be used to show where recovery and timeouts are needed. The reachability tree is then used to show deadlock freeness and consistency. Timing requirements are found by adding time to the transitions in the model.

4.4.1. Site failures

Firstly consider the asynchronous version of the extended 2 phase commit, this has been shown without failures in figure (4.6). Now introducing site failures as defined by Skeen gives the Petri net of figure (4.11) below. The extra states in the Petri net, d_1 and d_2, can be considered as decision states and do not write log files. If a failure happens in these states then upon recovery a w_1 and q_2 log file is read respectively, therefore the same failure transitions for w_1 and q_2 can be assigned to d_1 and d_2 respectively.

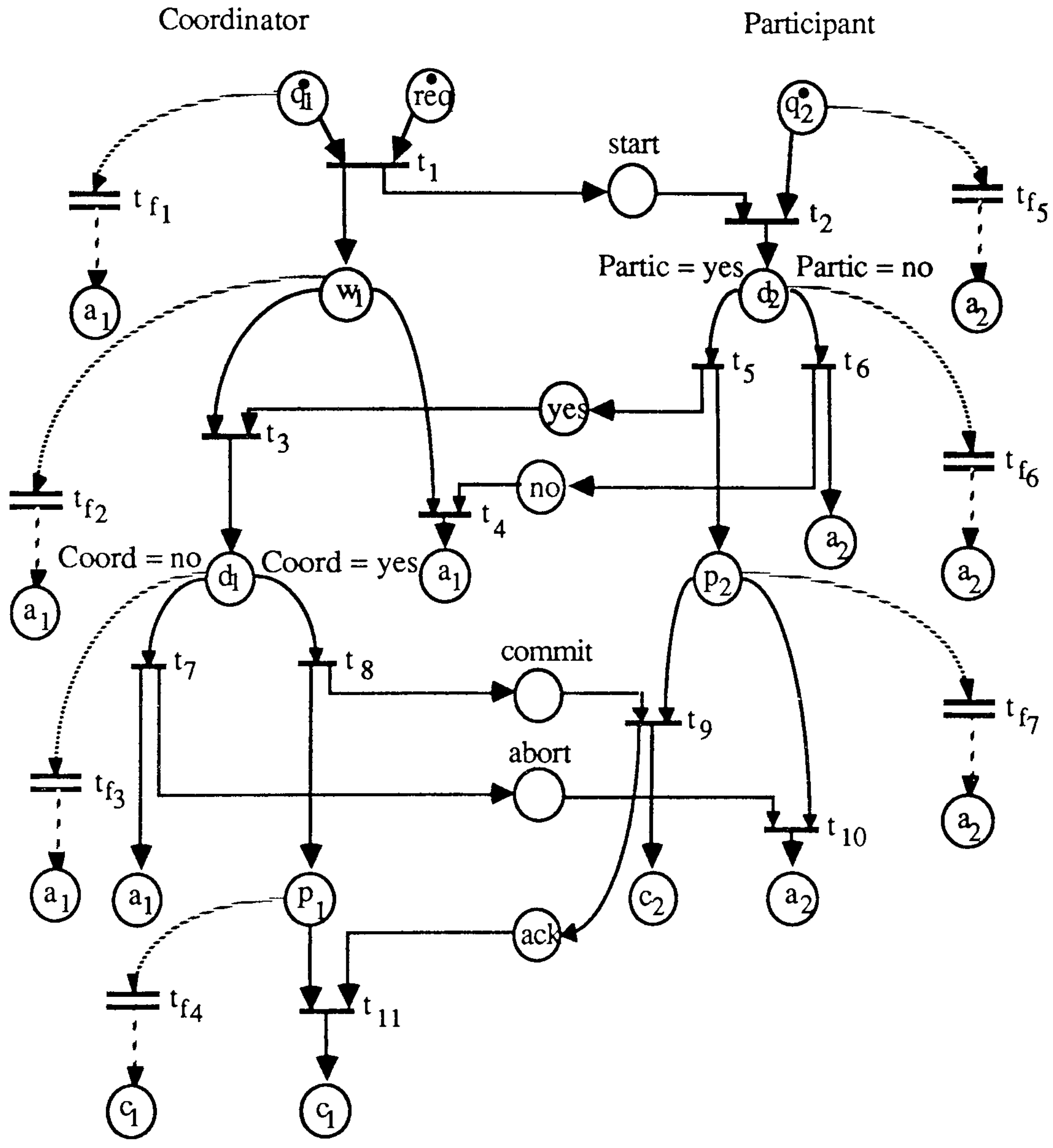


Fig.(4.11) Asynchronous extended 2 phase commit with site failures

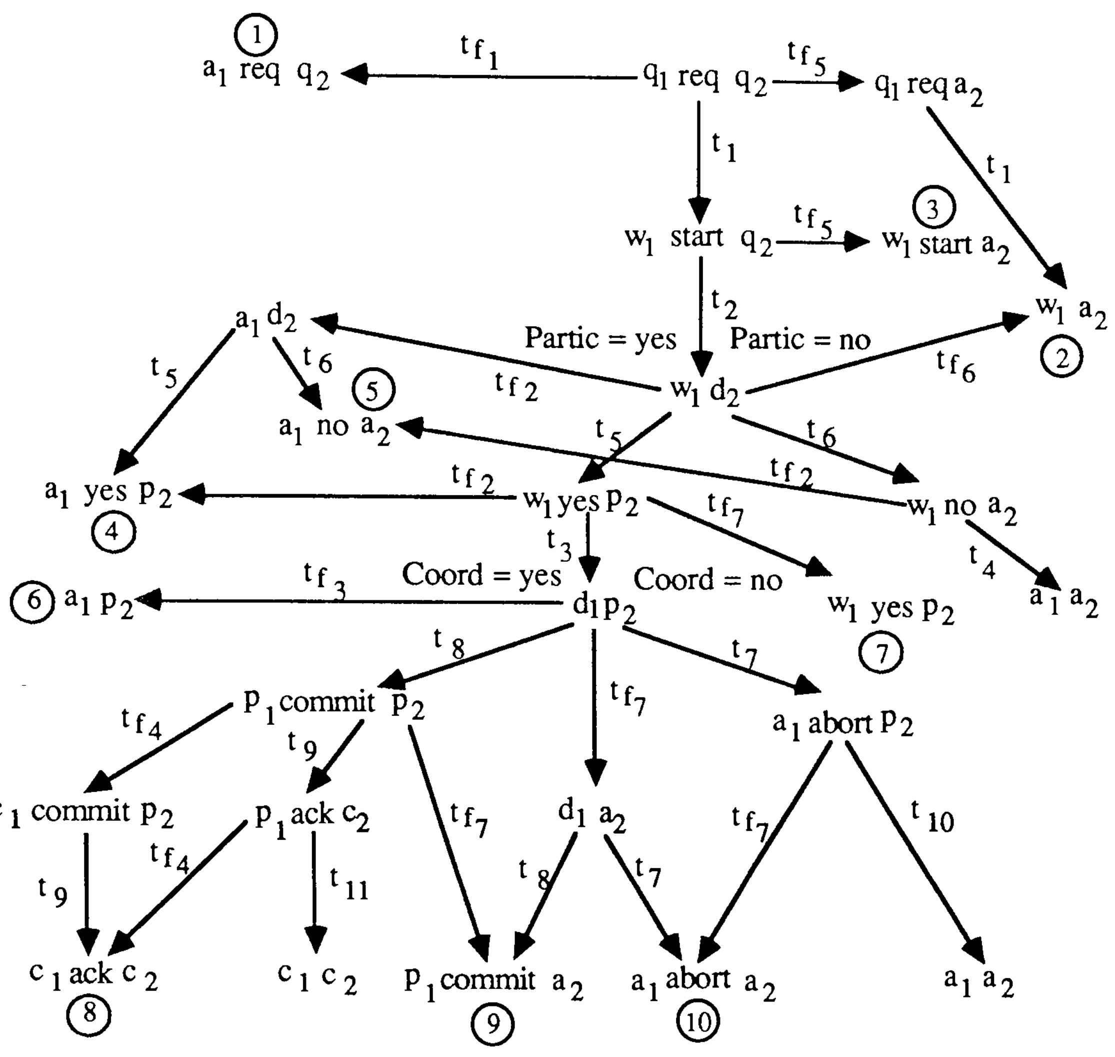


Fig.(4.12) reachability tree for fig.(4.11)

The reachability tree for figure (4.11) is shown in figure (4.12), this shows that with failures deadlock can occur in each of the circled states ((1) through (10)). To prevent this timeouts are assigned on states w_1, p_1, q_2 and p_2 as defined by Skeen (see section (3.4)). These timeouts are the same as defined by Skeen, thus this model and the analysis is equivalent to Skeen's FSM model.

The synchronous version of this protocol was presented without failures in figure (4.7) it is now shown with site failures in figure (4.13). The failures have been assigned in the same way as for the asynchronous version. However, Skeen's rule defining how these can be placed is not applicable because the concurrency set cannot be used since state transitions representing message transfer alter the state of two sites (as opposed to the asynchronous version where only one site changes state). The reachability tree for this is shown in figure (4.14) showing the deadlock states. This can be used to determine where timeouts should be included to prevent deadlock (cases (1) to (8)).

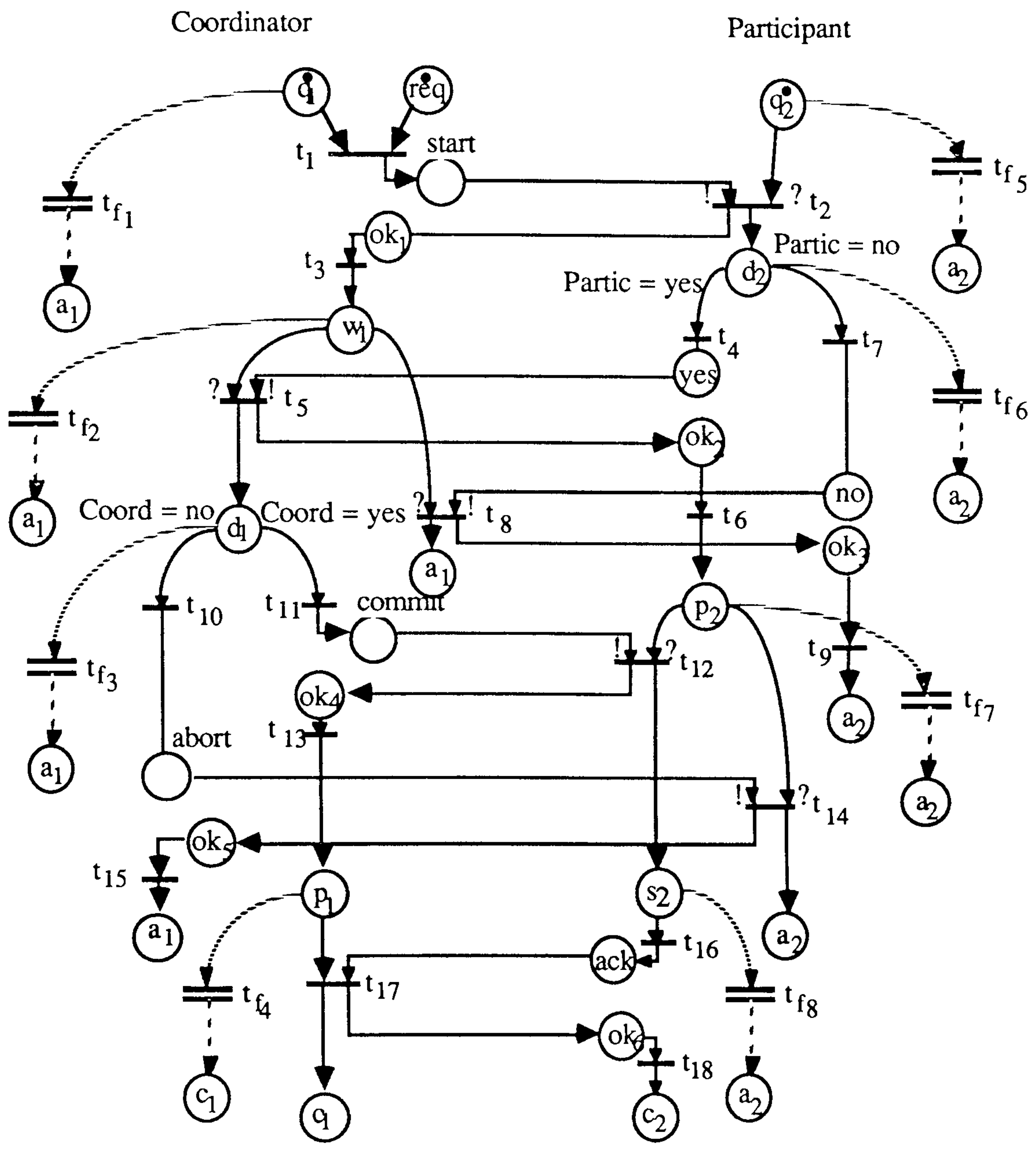


Fig.(4.13) Synchronous extended 2 phase commit with site failures

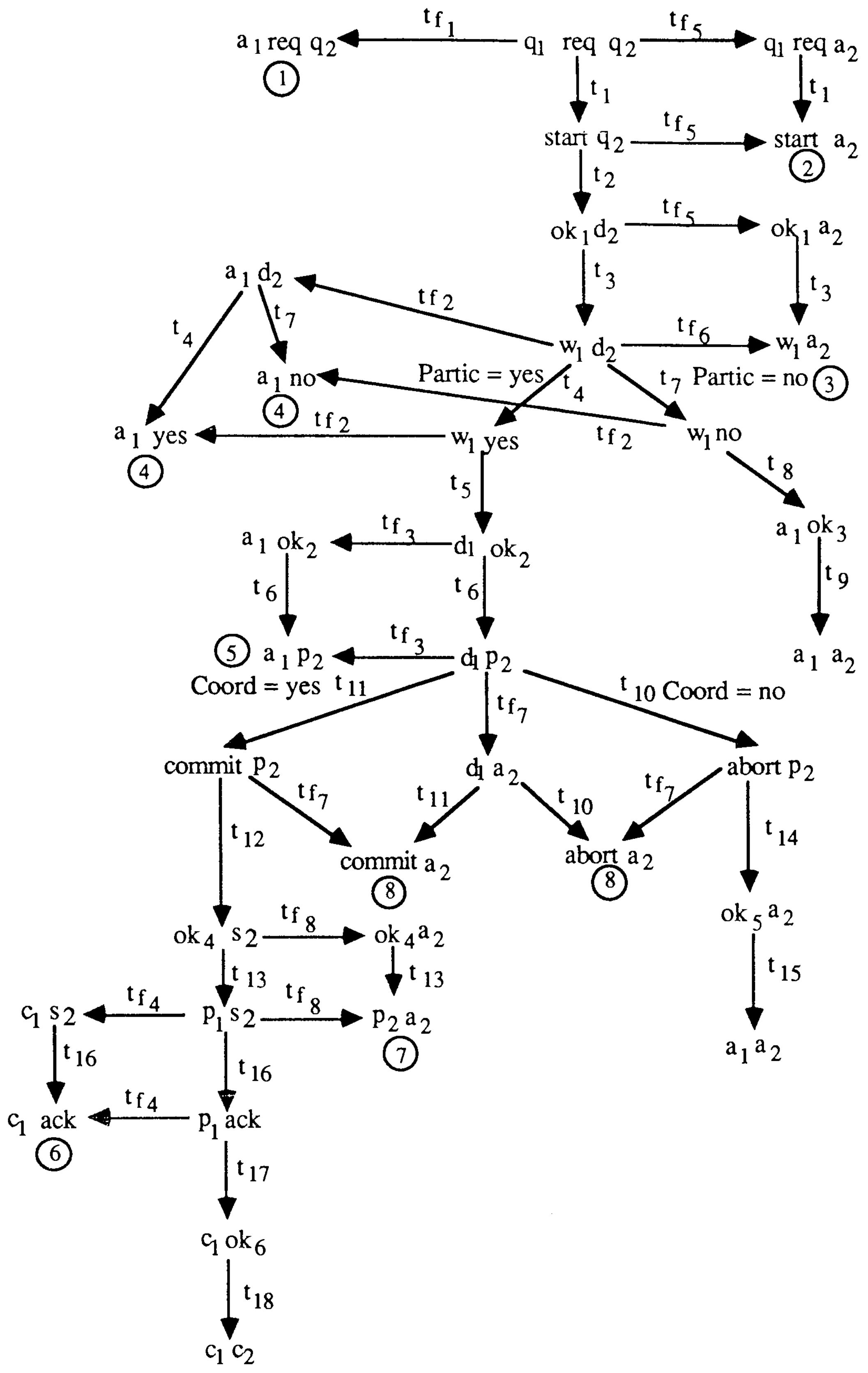


Fig.(4.14) Reduced reachability tree for fig.(4.13)

The reachability tree of figure (4.14) is again reduced, but this does not affect the analysis because the only states removed should never actually be reached. Also, recovery from the omitted states would be consistent because they do not cross a communication boundary. The timeouts are needed on both sides of every message transfer i.e. every message primitive must be bounded. This implies that a process must be able to stop waiting to receive a message and also stop trying to send a message. The limits on these bounds must be predetermined worst case values to prevent false timeouts. It must be noted here that a timeout applied to sending the ack message should transfer the state to c_2 thus it does not matter if the last ack is sent or not, this feature is used later for optimizing the protocol.

4.4.2. Link Failures

Link failure is more important than site failure because it is envisaged that link failure will prevail in a real-time control environment. In this thesis a tree structure is used for the protocols and so a single link failure is equivalent to network partitioning, this is a much more difficult situation to deal with than a single site failure [Davidson 85]. As pointed out before only protocols where messages are not lost are resilient to network partitioning, this is shown to be true here.

A link failure is assumed to be a physical disconnection of a communication link between processors, it is also assumed that messages are short so that a link disconnection either happens before or after a message has been received (and not half way through a message).

Consider the Petri net model of the asynchronous extended 2 phase commit, shown in figure (4.4), for the first message a link could be disconnected any time before t_2 fires or anytime after t_2 has fired.

If the start message has been received before the link becomes disconnected, the only problem is that the next message (yes/no) cannot be sent. However, if the link is disconnected before the start message is received it is equivalent to the transition t_2 being unable to fire. This is equivalent to losing a token from the start place. Thus a link failure can be assumed to be equivalent to a message loss in the asynchronous protocol. This is modelled in the Petri net as the removal of a token from a message place.

These lost message transitions can be added to the Petri net model of the asynchronous extended 2 phase commit as shown in figure (4.15). The reachability tree for this is shown in figure (4.16) and shows that there are six possible states after a link failure, namely, w_1q_2, w_1p_2, w_1a_2, a_1p_2, p_1p_2, p_1c_2.

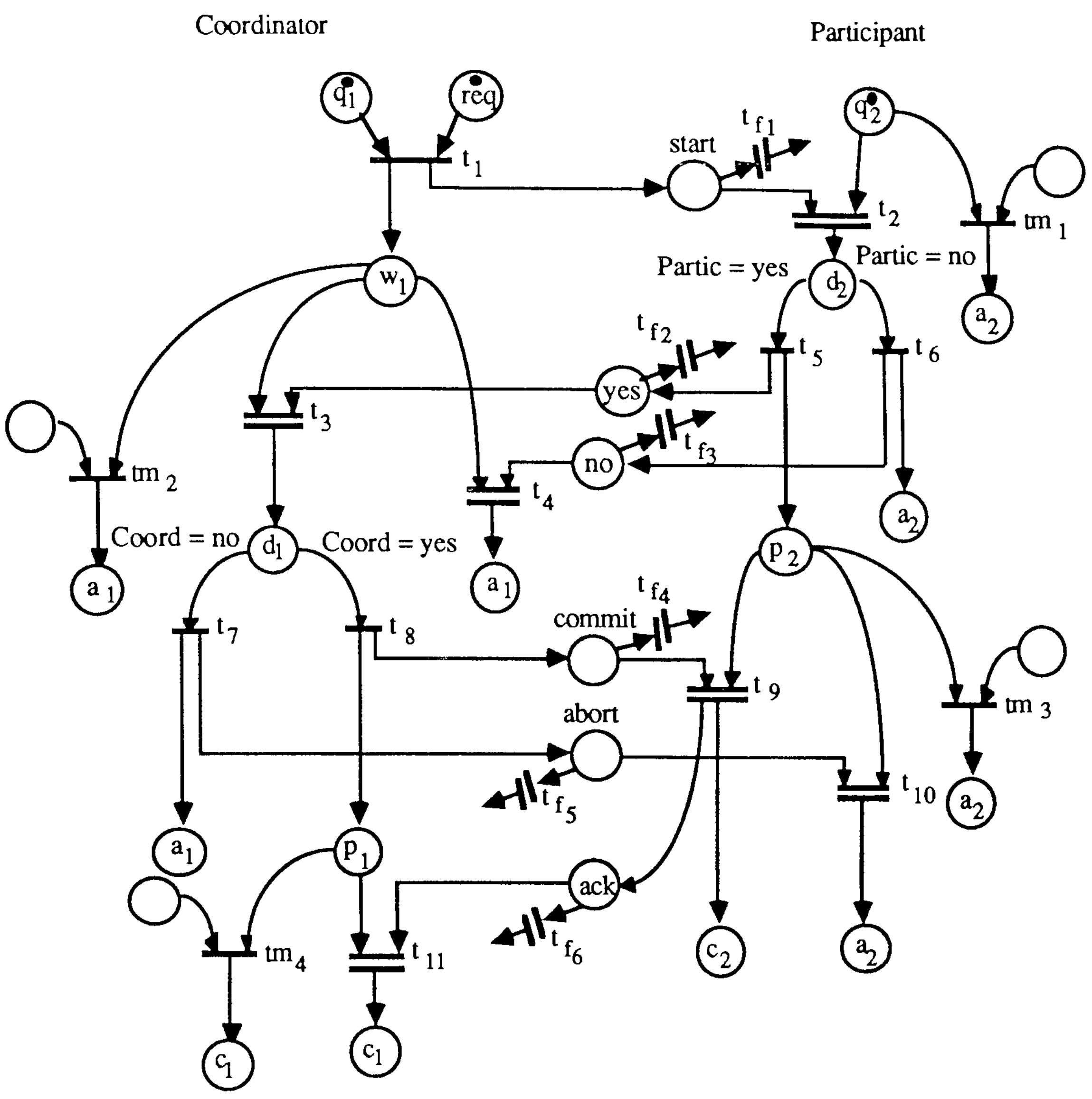


Fig.(4.15) Asynchronous extended 2 phase commit with link failures

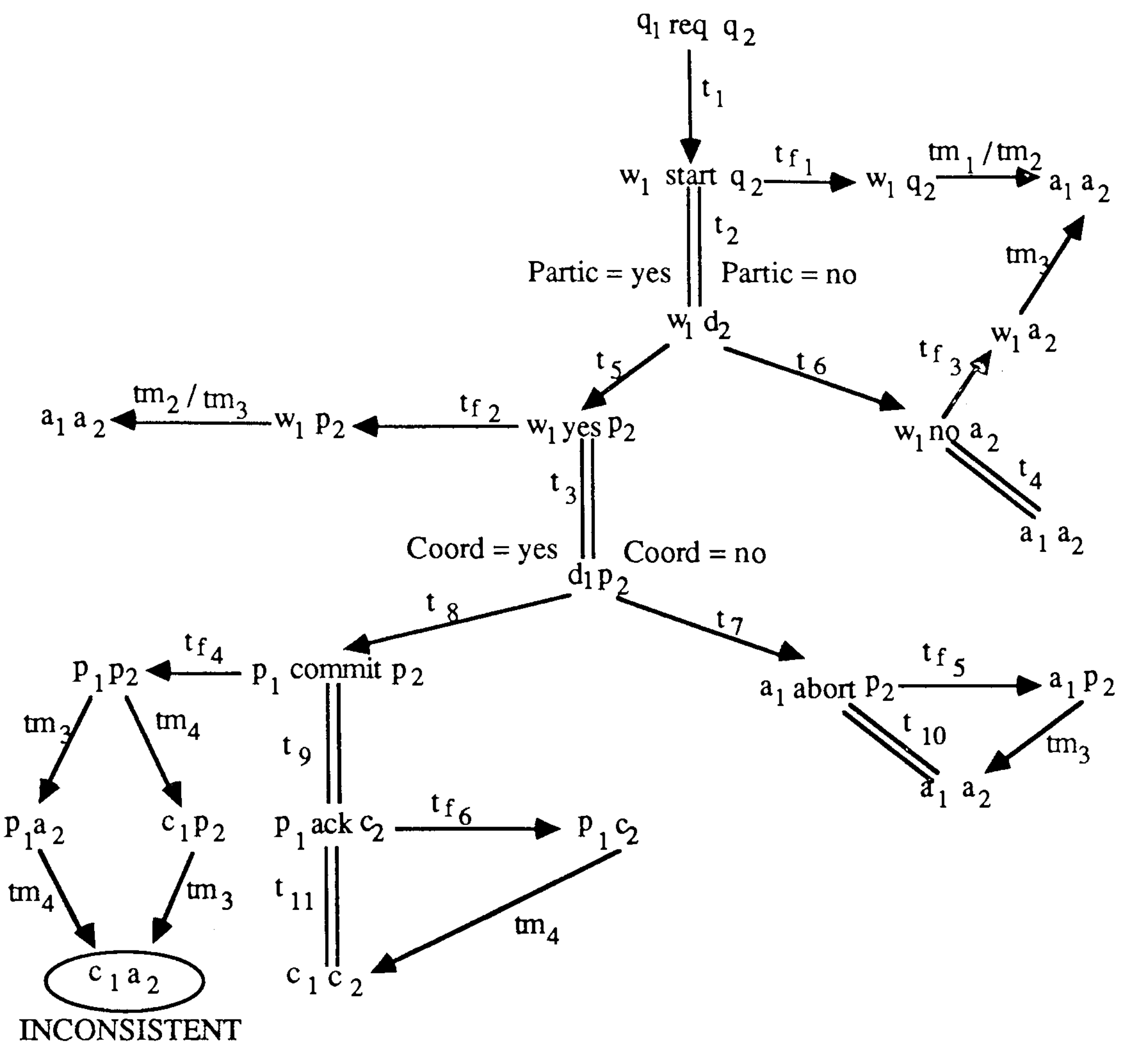


Fig.(4.16) Reachability tree for fig.(4.15)

From these states it can be determined that :-

(1) w_1 must timeout to a_1 (to be consistent in w_1 a_1)

(2) q_2 must timeout to a_2 (to be consistent in w_1q_2 with w_1 timing out)

(3) p_2 must timeout to a_2 (to be consistent in a_1p_2)

(4) p_1 must timeout to c_1 (to be consistent in p_1c_2)

Now if the state p_1p_2 is reached both sites timeout independently and an inconsistent final state is reached ie. c_1a_2. This shows that this asynchronous protocol is not resilient to message loss and that Skeen's rules are valid in this case.

A link failure in a system with synchronous communications can be modelled as a transition failure, i.e. a transition not firing, this is shown in the following diagrams as a double bar. With synchronous communications an acknowledgement is returned immediately after a message has been received thus a sender knows if a message was received or not. A link failure can now occur before the message is sent, after the message is sent but before the acknowledgement is returned, or after the acknowledgement. The latter case has no effect but both earlier cases have the same consequence in that no acknowledgement is received by the sender, this implies that the sender knows if a message was received or not.

If a message was sent correctly and then the link failed, the acknowledgement would fail to be sent and the sender site assumes that the message was not received and no action was taken by the receiver. The addition of possible link failures is shown in figure (4.17), double bars indicating where failures may occur. Timeouts have been included to show where they can be placed. The placement is derived from the reachability tree but shown here for conciseness. The protocols using synchronous communications have been implemented in Occam and shown to behave as expected, the code for these protocols with timeouts placed for link failure is included in Appendix B.

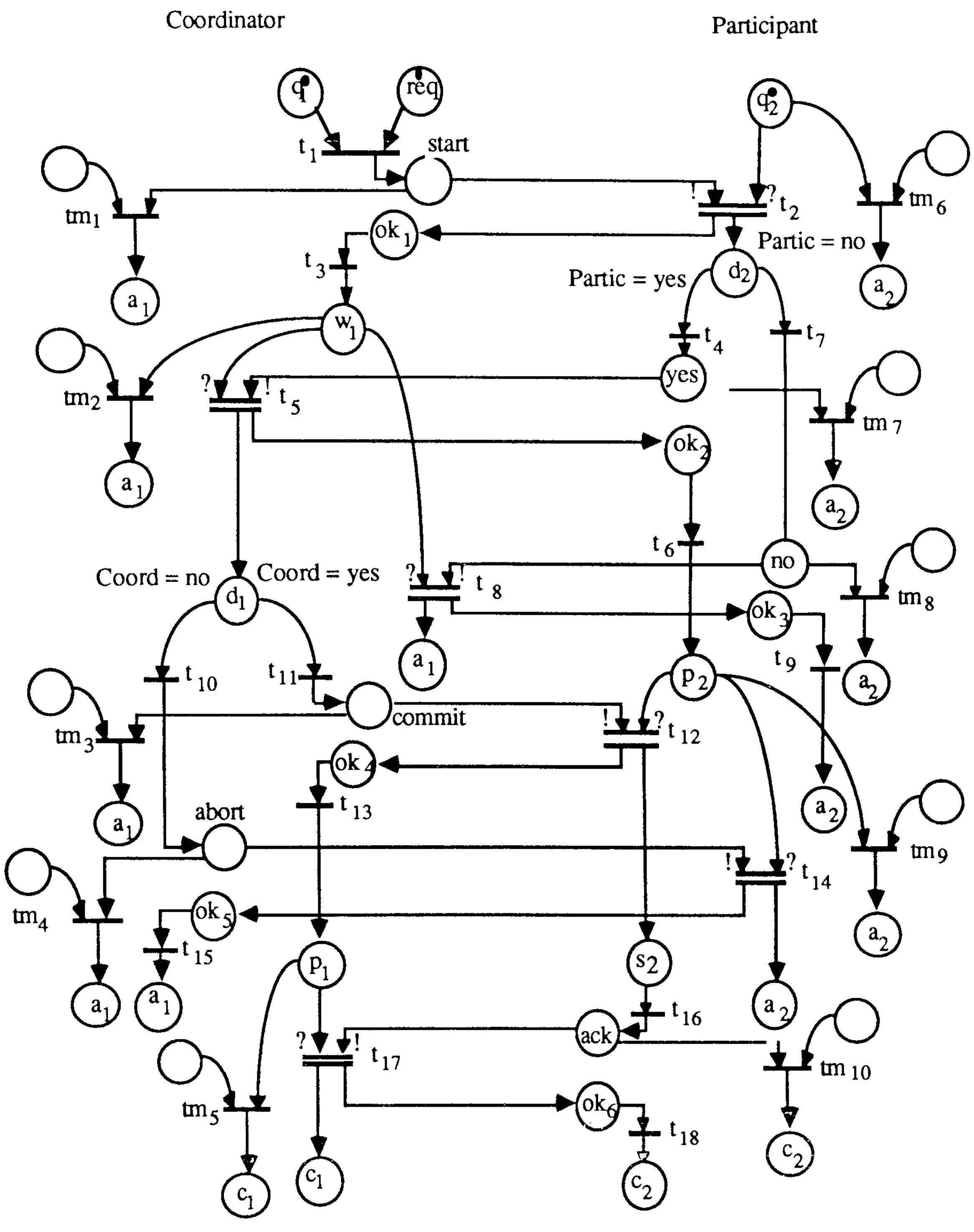


Fig.(4.17) Synchronous extended 2 phase commit with link failures

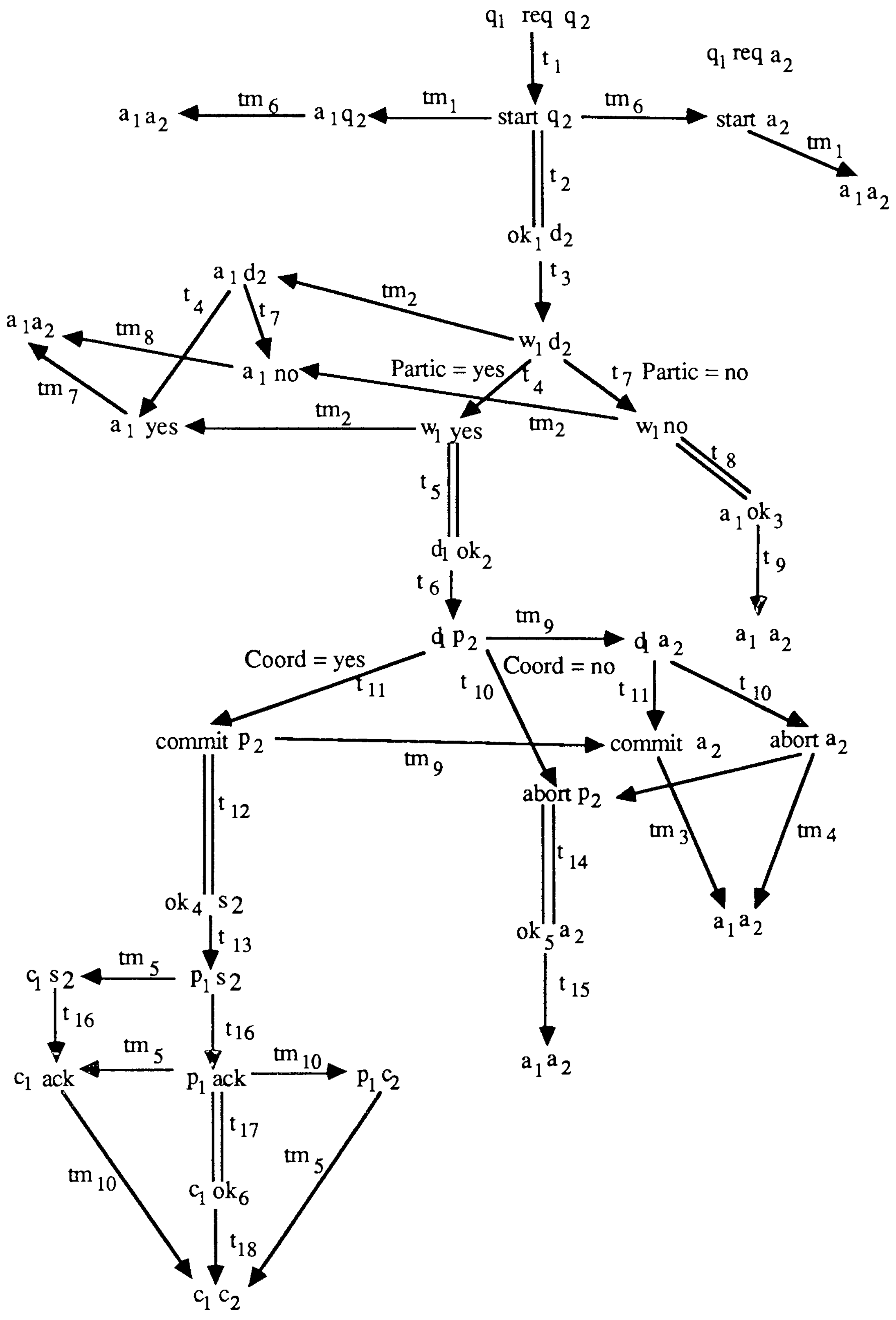


Fig.(4.18) Reduced reachability tree for fig.(4.17)

The reachability tree is shown in figure (4.18) the double transitions bars showing where a failure may occur. The states before these transitions show where timeouts must be used to prevent deadlock, such that the timeouts bring the sites to consistent states. It can be derived from the reachability tree that to ensure a link failure is tolerated every input and output must be bounded. The reachability tree shows that this two site protocol is resilient to a single link failure. The concurrency set from the reachability tree cannot be used to determine Skeen's rules but similarities between this model and Skeen's can be drawn. It must be noted that the timeouts on the places w_1, p_1 and p_2 are the same as for the asynchronous model, Skeen also defines 'undeliverable message' transitions which are equivalent to timeouts on sending a message, in this case a timeout on sending commit/abort and yes/no.

4.4.3. Timing analysis

An independently recoverable protocol seems well suited for use in a real-time environment since they have the capability of surviving both site and link failures. However, the recovery time of sites cannot be calculated and this makes timing analysis difficult, therefore only link failures are considered further. This section determines timing information relevant to the synchronous extended 2 phase commit in the presence of a link failure. It is assumed that both sites should reach a consistent agreement within a time limit T and that the action of both sites aborting is a safe action and is preferable to an inconsistent or late decision. A late decision is assumed to be unsatisfactory because other processes depend on the decision, therefore timeliness is of paramount importance. This section shows how to calculate and control timeouts to prevent false timeouts and considers the synchronous extended 2 phase commit protocol of figures (4.17) with possible link failures and timeouts included. Only a brief description of how to calculate timeout values is provided, section (6.4.2) shows the calculation in greater detail with reference to an application.

Each transition t_i has two times associated with it, the minimum and maximum firing times denoted $Min(t_i)$ and $Max(t_i)$ respectively. Therefore once enabled a transition fires in between the interval $[Min(t_i), Max(t_i)]$. Consider the events following the sending of the start message. To prevent a false timeout enough time must be allowed for t_2 to fire before either site can timeout, therefore $Min(tm_1)$ and $Min(tm_6) > Max(t_2)$. In this case $Min(t_2) = 0$ and $Max(tm_2)$ can be determined as the propagation delay of the start message plus the propagation delay of an acknowledgement. Also $Max(tm_1)$ and $Max(tm_6)$ must be less than the required deadline.

As an example consider a failure in which the link is disconnected before the first message start is sent, the latest an abort decision is reached is,

$MAX\ [(Max(t_1) + Max(tm_1), Max(tm_6)]$

The other timeouts are calculated in a similar fashion.

To see how timers may overlap consider the timing diagram shown in figure (4.19) which shows the messages and associated timers of the first phase of the extended two phase commit. The propagation delay of a message, Δ is assumed to be the same as for an acknowledgement because the messages are relatively short.

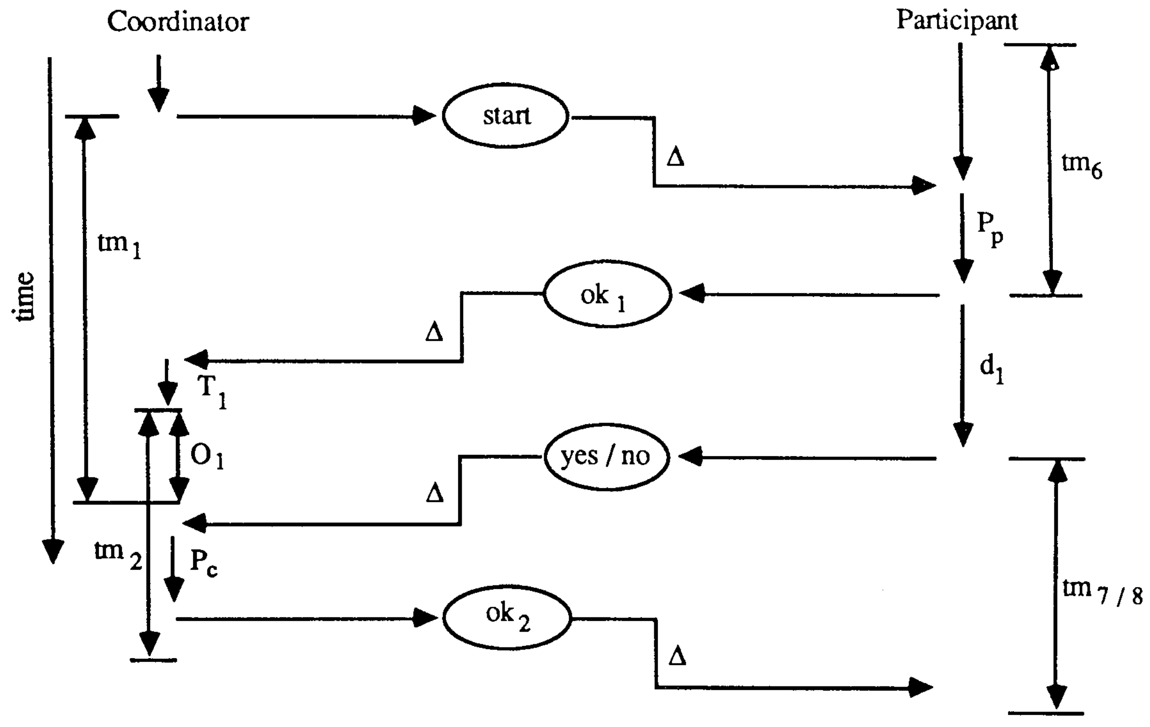


Fig.(4.19) Timing diagram of message transmission

From this diagram it is clear that timers may overlap, as shown by tm_1 and tm_2 but this does not cause problems because tm_2 is never started until after ok_1 is received.The processing delays P_p and P_c are included to show that messages take a finite time to process, to send an acknowledgement message this is negligible and could probably be omitted. The timer tm_1 is started when the start message is sent, if for some reason an acknowledgement is not received before it expires then recovery action is needed. If on the other hand the acknowledgement is received before tm_1 expires then after the time to process it another timer tm_2 is started.

A false timeout can occur if for some reason d_1 is longer than expected, in which case the yes/no message will be sent after tm_2 has timed out. Timer tm_7/tm_8 would then timeout even though no failure occurred. It must be ensured when calculating timeout values that worst case timing assumptions such as message propagation delay are used. The first timer, tm_1 in this case can have the approximate value of 2Δ where Δ is the worst case message propagation delay. Once the acknowledgement is received, after time T_1 the timer tm_2 is started, the value of this cannot be approximated to 2Δ because the participant site has to make a decision about its local ability to commit, this takes time d_l which is probably not negligible. The other timeout values can follow in a similar fashion.

4.5. Discussion

This chapter has shown how commit protocols can be extended with time to provide timely decisions in the presence of failures, one major contribution being the modelling of commit protocols using Time Petri nets. This model includes as much state information as previous modelling techniques and also includes time which is vital for real-time applications. Another advantage this method has over previous models is that the communication system is also modelled, this has been exploited to show the difference between implementations of commit protocols using asynchronous and synchronous communications.

The Time Petri net models in this chapter have also included an explicit model of the communication system. In particular, both the 2 phase commit and the extended 2 phase commit protocols have been modelled with asynchronous and synchronous communications.

It has been shown that the Petri net model of the 2 phase commit using asynchronous communication is equivalent to the original FSM model and also that the synchronous 2 phase commit protocol is still a blocking protocol. The synchronous 2 phase commit protocol has been implemented in Occam, the code listing is given in Appendix B, this implementation has been used to demonstrate the blocking property.

The extended 2 phase commit protocol has also been modelled with asynchronous and synchronous communications. Both models have been subjected to site and link failures and the placement of timeouts suggested to prevent deadlock and an inconsistent final state. It has been shown that the asynchronous extended 2 phase commit cannot be made independently recoverable from a link failure. However, it has also been shown that the same protocol implemented using synchronous communications can be made independently recoverable from the same kind of failure. The placement of timeouts to provide such recovery has also been suggested.

The implementation of the synchronous protocols using a programming language designed for distributed systems is described in the following chapter. The applications of the protocols developed here to real-time systems are considered in chapter 6.

Chapter 5

Implementation and optimization of commit protocols using synchronous communications

5.1. Introduction

The previous chapter has shown that Petri nets can be used to design commit protocols. It has shown how site and link failures can be modelled and used in the specification of robust protocols. These models can then be analysed so that recovery mechanisms can be designed and included to produce robust protocols and that timeouts can be used to detect failures. It has been shown why site and link failures can not be distinguished easily. The method was demonstrated through the design of a robust commit protocol that would survive a single link failure.

For the Petri net specifications and designs to be useful they must be easily transformed into implementations. Since the protocols were designed for distributed systems, if they are to be implemented successfully they must be programmed using a language suitable for expressing the properties of such systems. A suitable language needs constructs to describe parallelism, inter-process communication, synchronization and non-determinism. Occam is a language that possesses all the necessary properties quite neatly. Previous work has shown that Occam programs can be modelled by Petri nets [Carpenter 88], a reverse transformation is informally used in this chapter.

This chapter shows how Occam programs can be derived almost directly from the Petri net designs for synchronous communications shown in the previous chapter. This chapter demonstrates this translation process and emphasizes where care must be taken for the implementation to be practical. Also this chapter serves as an evaluation of the use of Occam in fault-tolerant distributed systems, and the limitations of Occam are discussed.

This work is novel in its implementation of robust commit protocols because point-to-point synchronous communications are used rather than the usual asynchronous network. This includes the use of the timed Petri net specification and designs and the use of structure preserving transformations to translate these designs into Occam programs. In addition, the optimizations of the protocol are novel in that the acknowledgment signals are used to convey information and timeouts are used in the decision process. An acknowledgement that is received designates a positive vote whilst not receiving an acknowledgement is assumed negative.

Section (5.2) describes the concurrent programming language Occam and compares it with Ada. The advantage that Occam is based upon a mathematical background is emphasized. The actual implementation of the commit protocols in Occam 2 is outlined in section (5.3).

From the implementation the robustness of the protocols to site and link failures can be tested, this is carried out by simulating the failures as outlined. It is shown that Occam has insufficient semantics to provide complete resiliency to all types of failures. A solution to this is presented using available assembler routines [Shepherd 87]. Section (5.4) explains how the commit protocols developed in chapter 4 with synchronous communications can be optimized. The optimizations reduce the number of messages sent by using the acknowledgement message to contain information. Using these optimizations a three site protocol resilient to a single link failure is developed and tested. Finally conclusions are summarized in section (5.5).

5.2. Occam as a language for distributed systems

Occam was developed by Inmos as the language for programming the Transputer [May 85]. Until recently it was the only programming language available for the Transputer. It has a sound mathematical background because it is based upon CSP [Hoare 78]. This allows simple transformations to be applied to programs with full confidence that the resulting program is equivalent to the original. Occam is a very simple language and contains all the primitives required to program distributed systems. The original Occam language has now been superseded by Occam 2, this contains the same core as the original Occam but includes extra features. When used in this thesis 'Occam' refers to the basic features which are present in both Occam 1 and Occam 2. The programs in this thesis have been developed using Occam 2 but throughout the description of the software Occam and Occam 2 will be used interchangeably.

5.2.1. Occam 1

This section describes the original definition of Occam. The later variant Occam 2 contains the same primitives and constructors as Occam 1, the differences between Occam 1 and Occam 2 are considered in section (5.2.1). Occam 1 is a block structured language but unlike conventional languages the indentation of the program is also part of the syntax, programs must therefore be carefully formatted. Like most other languages Occam 1 consists of primitives, constructors and declarations, but unlike many languages these constructs contain the necessary commands needed to program distributed systems.

Occam 1 primitives are assignment (:=), communication input (?), communication output (!), process halt (STOP) and a null process (SKIP). The input and output commands provide synchronized inter-process communication which is restricted to being one-to-one along predefined channels. Since the communication is synchronized it can be used to provide synchronization of processes. An example of two processes communicating is shown below :-

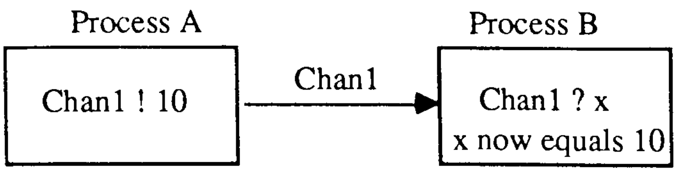


Fig.(5.1) Two communicating Occam processes

The other properties required by a distributed system programming language, parallelism and non-determinism are provided by the use of constructors. The constructors of Occam are SEQ, PAR, IF, WHILE, ALT. The first two, SEQ and PAR describe how the set of commands which follow are to be executed, either sequentially or in parallel. When PAR is used the processes which follow can be executed concurrently either on a network of Transputers or on one Transputer. When more than one process is run on one Transputer the processor is time sliced using a micro-coded scheduler. Examples of the use of the sequence and parallel constructors are given below :

```
SEQ                       PAR
  x :=  b                   x := b
  y := c                    y := c
  process1                  process1
```

Although good for expressing the parallelism of a system PAR incurs an overhead if the processes are executed on one Transputer (context switching time of 75 nS). When using the parallel construct certain parallel programming rules must be observed such as all the variables must be mutually exclusive [Dijkstra 68].

The IF and WHILE constructors provide selection and repetition which are amongst the basic requirements of a sequential programming language [Dijkstra 72], whilst the ALT (alternation) provides non-determinism. An ALT allows one of a number of guarded commands to be chosen non-deterministically. A guarded command [Dijkstra 75] consists of a guard followed by a number of commands to be executed if the guard is chosen. A guard may consist of an optional boolean expression and an input command, timer input or a SKIP command. Timer inputs are used to break out from communication and prevent deadlock. An example of an ALT command is :

```
ALT
  C1 ? x
    B1 ! x
  (A AND B) & C2 ? y
    B2 ! y
```

When executed, the ALT waits until one of the guarded commands is ready to be executed. In the above example the first guard is ready when the corresponding output on channel C1 is ready, the second guard is ready if the expression (A AND B) is TRUE and the output on channel C2 is ready. Once a guard is chosen, the defined actions are then performed. If more than one guard is ready when the ALT is executed an arbitrary guard is chosen. However, a PRI ALT construct exists which allows priorities to be assigned to each guard.

This means that if a PRI ALT is executed and a number of guards are ready then the first one in textual order is selected. One restriction on these constructs is that only input commands are allowed in the guard.

The declarations allowed in Occam are CHAN (channel), VAR (variable), DEF (constant), VALUE (fixed procedure parameter) and PROC (procedure). The declarations of CHAN, VAR and VALUE may also be subscripted to allow for arrays. A PROC declaration is used for convenience to bind a set of primitives, constructors and declarations, it does not increase the expressiveness of the language.

Being a very simple language Occam 1 was criticized for not including features which had become expected of modern programming languages. Notably missing were data typing, recursion, functions and modules. Most languages allow variables to be declared as integers, boolean or real but Occam 1 only allows integer variables. Recursion is also not allowed in Occam 1 due to the problems of implementing large stacks on the Transputer. Functions are not provided but with careful programming side-effect free procedures can be used as functions. Occam 1 does not allow modules which are crucial for any large software project but separate compilation of procedures is provided by certain programming environments [Inmos 89]. Another problem with Occam 1 is the use of a predefined read-only channel, TIME, to act as a timer, this channel breaks the rules of Occam in that it is not one-to-one and can be accessed by many processes. This means procedures using this channel cannot be transformed using the usual rules. Another problem is the type of communication used is such that only one data item can be transferred but often it is required to send a complex data structure per rendezvous.

5.2.2. Occam 2

Occam 2 was developed in an attempt to solve the problems of Occam 1. It allows variables to be typed, INT, BYTE or BOOL and thus provides type checking. However it still does not provide other useful data types such as user defined types (eg. records, queues etc). Another type included is the TIMER which defines a channel which can only be used to input from a hardware clock.

```
TIMER clock :
clock ? time.now
```

These timers allow a representation of 'computer time' and not 'real-time' and also follow the traditional Occam model of one-to-one channels.

In Occam 2 a channel is now declared as having a type so that they can be checked by the compiler for misuse. As an example :

```
CHAN OF INT chan1 :
```

Any attempt to send a variable of type BOOL or BYTE would result in an error. Channels are now allowed to send complex data structures (Eg. Arrays) in one rendezvous.

Communication can also be made safer by the use of protocols which define what types and the order of data which can be sent / received along a channel. Any abnormalities can then be reported by the compiler.

Although included in the definition for Occam 2 functions and the CASE statement are not fully implemented by all the available compilers. Another problem, in particular for industrial credibility is that Occam 2 still does not allow any kind of module concept, only the external programming environments provide such support.

5.2.3. Other languages for programming distributed systems

Many concurrent languages exist but most are limited to research institutions. An excellent survey of almost all applicable languages is provided by Bal et al [Bal 90]. The most commercially available programming languages for distributed systems Ada [Dod 83] and Occam are considered further. This section contrasts the main distributed features of Ada and Occam.

Parallelism is expressed in Ada by the use of 'tasks' which can be created dynamically. Unlike Occam where statements can be executed concurrently, tasks are the only unit of parallelism in Ada. As an example of executing two processes, Proc1 and Proc2 in parallel :

```
Occam 2                  Ada
  PAR                    declare
    Proc1()                task one
    Proc2()                task two
                           task body one is
                             begin
                               Proc1
                             end one;
                           task body two is
                             begin
                               Proc2
                             end two;
                         begin
                           null
                         end ;
```

One disadvantage of the Ada type of parallelism is that the tasks cannot be passed parameters, they must be communicated separately. Occam on the other hand allows parameters in the procedure calls which can be executed in parallel.

Inter-process communication in Ada is known as an extended rendezvous. It is a form of synchronous communication but involves more waiting than that of Occam. The type of communication used by Occam (? and !) involves the sender process waiting until the receiver has acknowledged receipt, after this the sender continues independently. The extended rendezvous of Ada on the other hand involves the sender waiting until the receiver has received the message, performed some computation and then returned a reply. This has the disadvantage that concurrency is reduced.

The Ada form of communication is also known as remote invocation and is not to be confused with the similar remote procedure call. Syntactically they can be made to look the same but semantically they are very different because a remote procedure call transfers control not messages and does not need to have to execute an explicit receive command. An Ada send and receive command is outlined below :

```
Send                         Receive
.                            task Procl is
.                              entry send (num : integer);
Procl.send(x)                end Procl ;
.
.                            task body Procl is
                             begin
                               ...
                               accept send (num : integer)
                               ...
                             end Procl;
```

It can be seen that the send command is a form of a procedure call and the receive consists of an entry and an accept command. This is not as simple as the Occam model and also causes problems because it is not one-to-one, i.e. a number of tasks may send to the same receiver task.

Another problem with Ada is that communication can also use shared variables which although efficient is undesirable because of the lack of control and synchronization. This type of communication is not allowed in Occam. The communication used in Occam is part of the Occam model (communications are primitives) and is simpler and more predictable which makes it safer for use in critical real-time systems.

The final property required for distributed systems, non-determinism, is achieved in Ada using the 'select' statement which is similar to the Occam ALT. For instance :

```
select
  accept call_1 (...)
  ...
  end call_1
  ...
or
  accept call_2 (...)
  ...
  end call_2
  ...
or
  delay 0.5 seconds
  ...
end select ;
```

This allows communication with either branch depending on which one is ready first. If no rendezvous is taken within 0.5 seconds then the delay branch is selected and the commands following the delay statement are executed. If an accept is taken then the commands within the rendezvous are executed followed by any commands after the end of the accept. One advantage of the select statement is that it allows a procedure call in one of the branches. Using this, a message can be sent and also timed, this is known as 'timed entry call'. It is restricted so that only one call is allowed within the select command and accept statements cannot be used as well. The timeout period is for the call to be accepted and NOT the completion of the rendezvous which would probably be more useful.

Occam is more suitable than Ada for implementing commit protocols because it allows a direct form of synchronous communication and a simpler representation of concurrency. Since each Occam construct has a Petri net representation it is also easier to transform the Petri net specification into Occam code. It would be possible to implement the Petri net specifications in Ada but problems may arise due to the form of Ada communications. It would also be necessary to develop Petri net representations for Ada constructs.

5.2.4. Petri net models of Occam constructs

Petri nets have been shown to be a powerful modelling technique (chapter 4). For the Petri net specification of a protocol to be transformed into Occam, it must be shown that Occam constructs can be modelled using Petri nets. These Petri net structures can then be identified within the specifications and transformed into Occam.

Previous work on the modelling of Occam using Petri nets [Carpenter 87, Carpenter 87b, Carpenter 88] has developed a set of transformations. The Petri nets of typical Occam constructs are shown in figure (5.2).

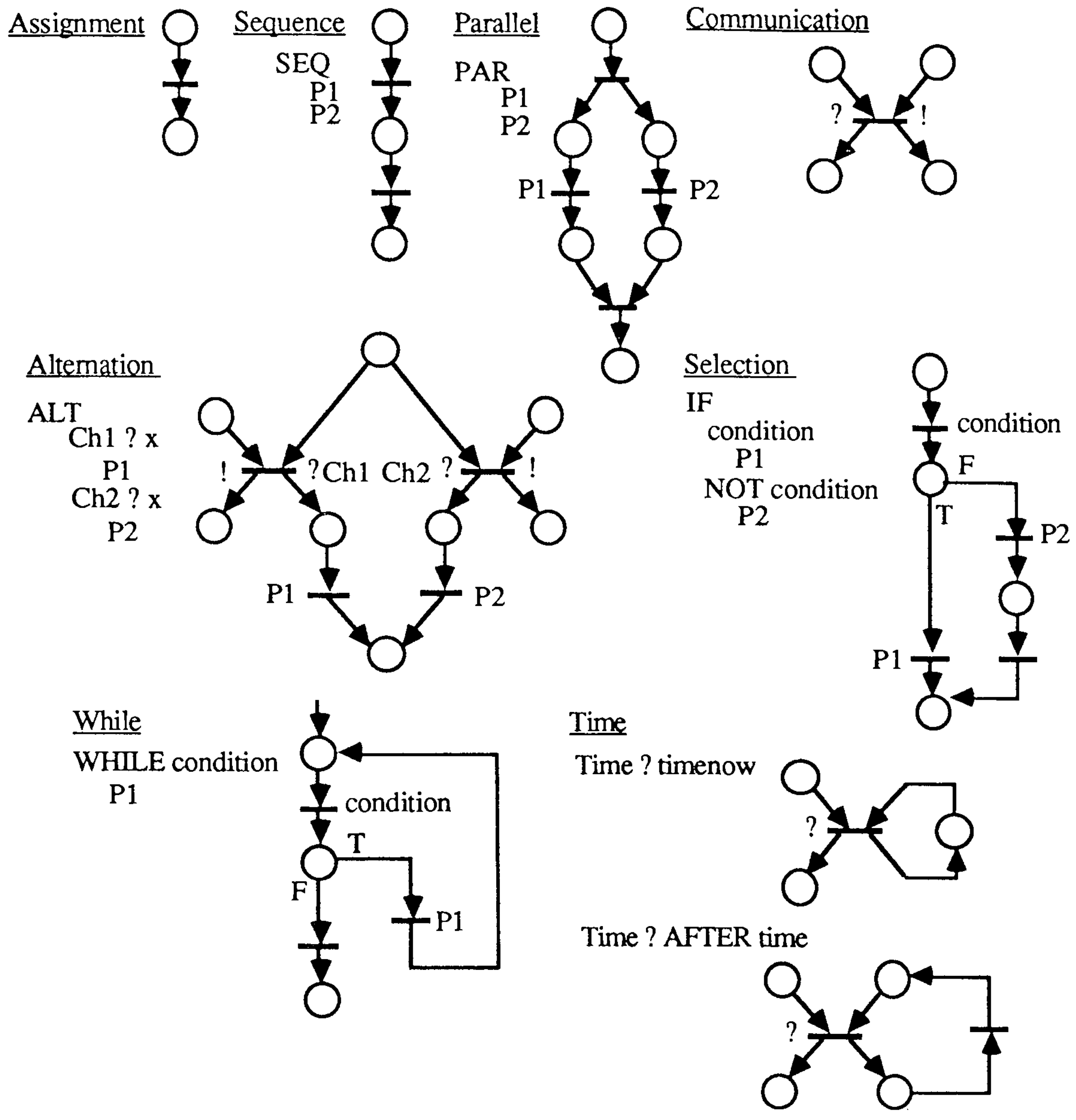


Fig.(5.2) Petri net representations of Occam constructs

These show the control flow of the constructs, for example the parallel construct will only terminate when both constituent parts have completed, i.e. P_1 and P_2 both finished. An ALT or an IF statement will terminate when the selected statements have completed. This chapter uses the above Petri net constructs to implement the protocols in Occam.

5.3. Occam implementation of commit protocols

Occam has in-built primitives to provide inter-process communication, namely ? and ! for input and output respectively. The communication is synchronous and should map directly into the previously described synchronous specifications. The idea of time is also included in the language which allows it to be used in real-time systems [Leppalla 87]. This section outlines how the synchronous specifications of a commit protocol can be almost directly mapped into Occam. For demonstration purposes, the implementation of the extended 2 phase commit is explained in detail. It is then shown how failures can be modelled in Occam and timeouts provided to prevent deadlock.

At the top level for a two site protocol, an Occam model of the specification is formed using a parallel construct :-

```
PAR
  coordinator (To.participant, From.participant)
  participant (From.participant, To.participant)
```

this allows two processes, the coordinator and participant to execute concurrently and pass messages to and from each other. This construct can be used on a single Transputer or each process can be run on a separate Transputer. The protocols in this thesis were developed using a multi-Transputer system where each Transputer was used for one process only.

Consider the specification without any failures as shown in figure (4.5). The message places ($start_1$, ok_1, etc.) can be implemented as messages sent along channels or links and each state place is equivalent to a state assignment. The decision states (d_1, d_2) where each site makes a local decision whether to commit/abort, can be modelled for simulation purposes using a boolean variable which is set to TRUE/FALSE.

Also each transition marked ? and ! can be translated into an Occam communication pair. One of the problems here is that each communication transition represents a separate channel. For practical reasons the number of actual channels was reduced to two, namely To.participant and From.participant. This means that the messages yes and no are sent along the same link (From.participant), the same applies to the messages abort/commit. This restriction forces the code to perform a check on the message once received.

With this restriction the protocol cannot be made independently recoverable to site failures but resiliency to link failures is still possible. Site failures need extra communication upon recovery because a site may fail after a commit message is received but before the commit action (and therefore log file) was performed, upon recovering the failed site will wait for the decision from the coordinator. To solve this a reasonable assumption would be that a site never fails between receiving a message and writing a log record. This could cause deadlock if the coordinator assumes that the participant has finished because it will not send the decision again. Link failures are unaffected because each site can still continue processing. The outline for an Occam implementation of the 2 site protocol where failures are not allowed is as follows :-

```
Proc coordinator (To.p1, From.p1)

  SEQ
    state := $q_1$
    To.p1 ! start
    state := $w_1$
    From.p1 ? mess
    IF
      mess = no
        state := $a_1$
      mess = yes
        IF
          coord.willing
            SEQ
              -- decision is commit
              To.p1 ! commit
              state := $p_1$
              From.p1 ? mess
              IF
                mess = ack
                  state := $c_1$
                TRUE
                  -- error
          NOT coord.willing
            SEQ
              -- decision is abort
              To.p1 ! abort
              state := $a_1$
:

PROC participant (To.coord, From.coord)

SEQ
    state := $q_2$
    From.coord ? mess
    IF
      mess = start
        IF
          partic.willing
            SEQ
              To.coord ! yes
              state := $p_2$
              From.coord ? mess
              IF
                mess = commit
                  SEQ
                    To.coord ! ack
                    state := $c_2$
                mess = abort
                  state := $a_2$
          NOT partic.willing
            SEQ
              To.coord ! no
              state := $a_2$
:
```

As described above, using a single link prevents protocols being independently recoverable to site failures so only link failures will be discussed further. A link failure is effected by actually removing the physical link between Transputers during simulation. As mentioned in a previous section (4.4.2) to be resilient to a link failure each input and output process must be bounded.

It has been shown [Carpenter 87] that Occam allows an input process to be bounded by the use of an ALT statement and a timer process as shown below :-

```
time ? time.now
ALT
  input ? message
    . . . process input
  time ? AFTER time.now PLUS allowed.wait
    . . . timed out
```

When the ALT statement starts either the input process or time process can take control, the time process will take control only after the time specified thus allowing the input process that amount of time to communicate. This provides a method of bounding an input process.

It would be expected that the output process could be dealt with in a similar manner but because of language limitations output guards are not allowed in an ALT statement and can not be bounded in this way. A method to allow indeterminacy in input/output guards has been proposed [Bornat 86] but timeouts can not be implemented in this way. Therefore pure Occam cannot be used to implement protocols resilient to failures.

Fortunately, if the protocols are implemented on Transputers, assembler routines have been provided which allow bounded inputs and outputs [Shepherd 87]. These routines still provide synchronous communications but provide a way of breaking out and preventing deadlock. Procedure calls are used to replace the communication primitives ? and ! and are of the form :-

```
InputOrFail.t   (Channel, message to be sent, timer, allowed
delay, flag)
OutputOrFail.t  (Channel, message received, timer, allowed
delay, flag)
```

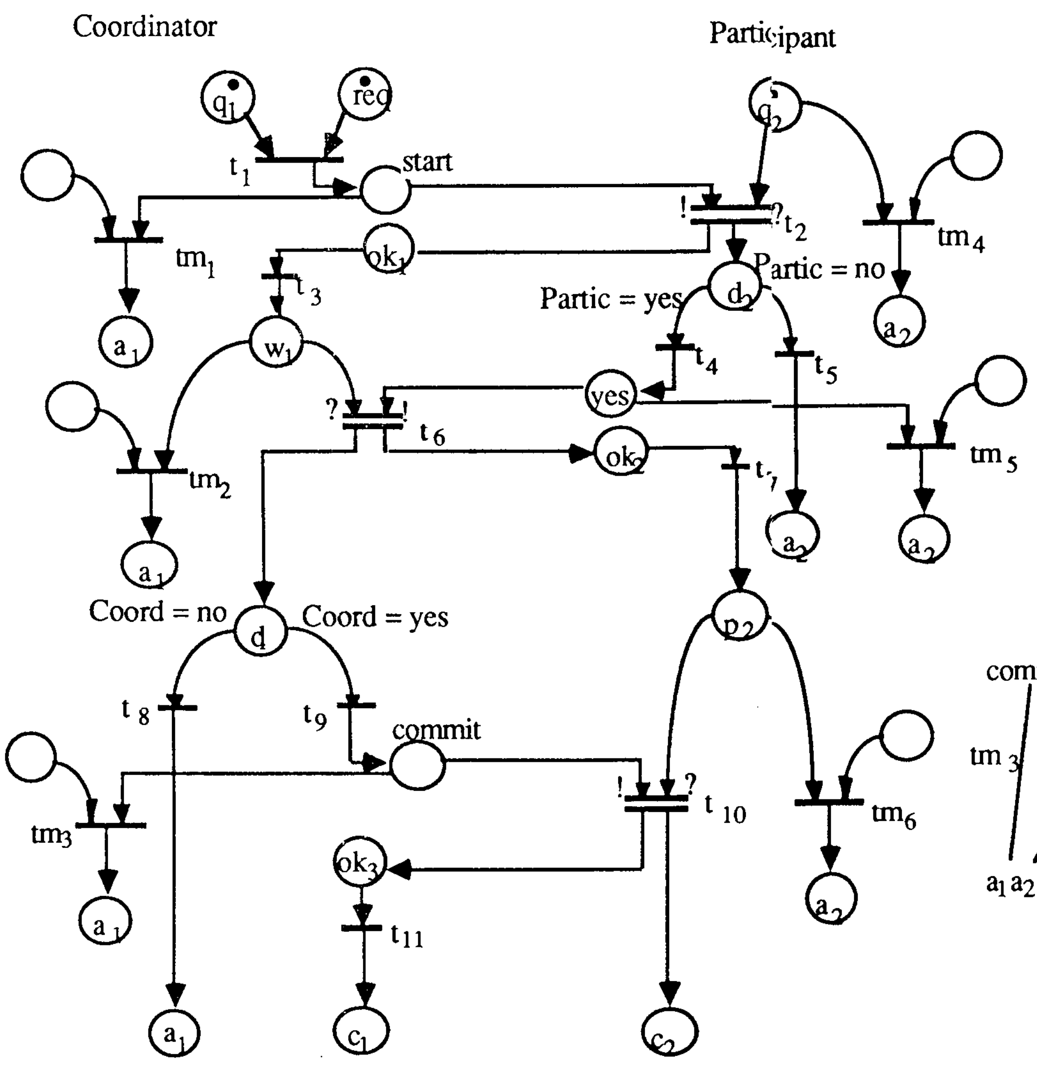
The flag is set to TRUE if the process has not completed within the allowed time else it is set FALSE. Each procedure call models exactly one side of the communications of figure (4.10b) and must be used to prevent deadlock.

The 2 phase commit and extended 2 phase commit protocols have been implemented using these routines to provide resiliency to site and link failures. The results of the implementations show that the 2 phase commit protocol remains blocking whilst the synchronous extended 2 phase commit can be made to terminate consistently. The listing of the implementation using these assembler routines is given in Appendix B.

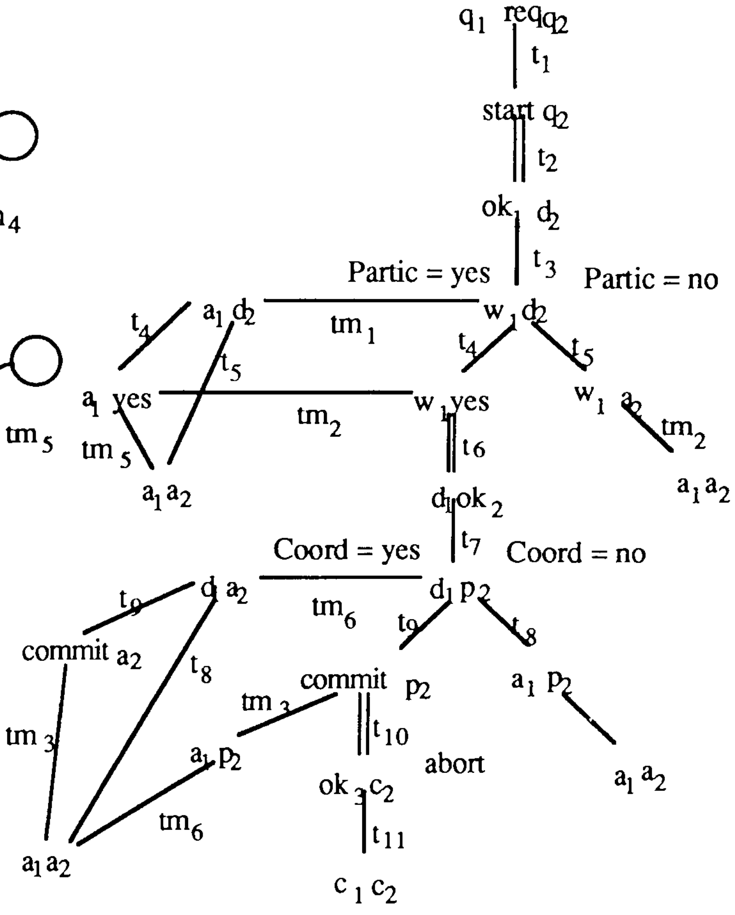
5.4. Optimizations with synchronous communications

This section proposes optimizations to the robust extended 2 phase commit using synchronous communications when only link failures are expected to occur. The full listing of the Occam code for the resulting protocol is included in Appendix B. The protocol specification shown in figure (4.17) is the basis for the optimizations, the initial observation being that if the abort command was not sent then both sites would timeout and abort which is the same situation if the abort command had been sent. This message therefore contains no additional information and can be removed.

The robust extended 2 phase commit consists of ten messages, five of which are acknowledgements. It would be preferable to reduce this number which does seem a bit excessive. For the two site protocol of figure (4.17) not only is the abort command message redundant but also the 'no' vote and 'ack' message. The 'no' vote can be removed and the timeout on w_1 used so that a timeout is assumed to mean a negative answer, i.e. the 'no' message. The same reasoning applies to the abort and ack messages. This reduces the number of explicit synchronous messages from five to three. The resulting protocol is that shown in figure (5.3). The reachability tree (figure (5.3b)) is again of the reduced form because the acknowledgement transitions (t_3, t_7 and t_{11}) are assumed to fire as quickly as possible.



(a) Petri net



(b) Reduced reachability tree

Fig.(5.3) Extended 2 phase commit - optimized for synchronous communications

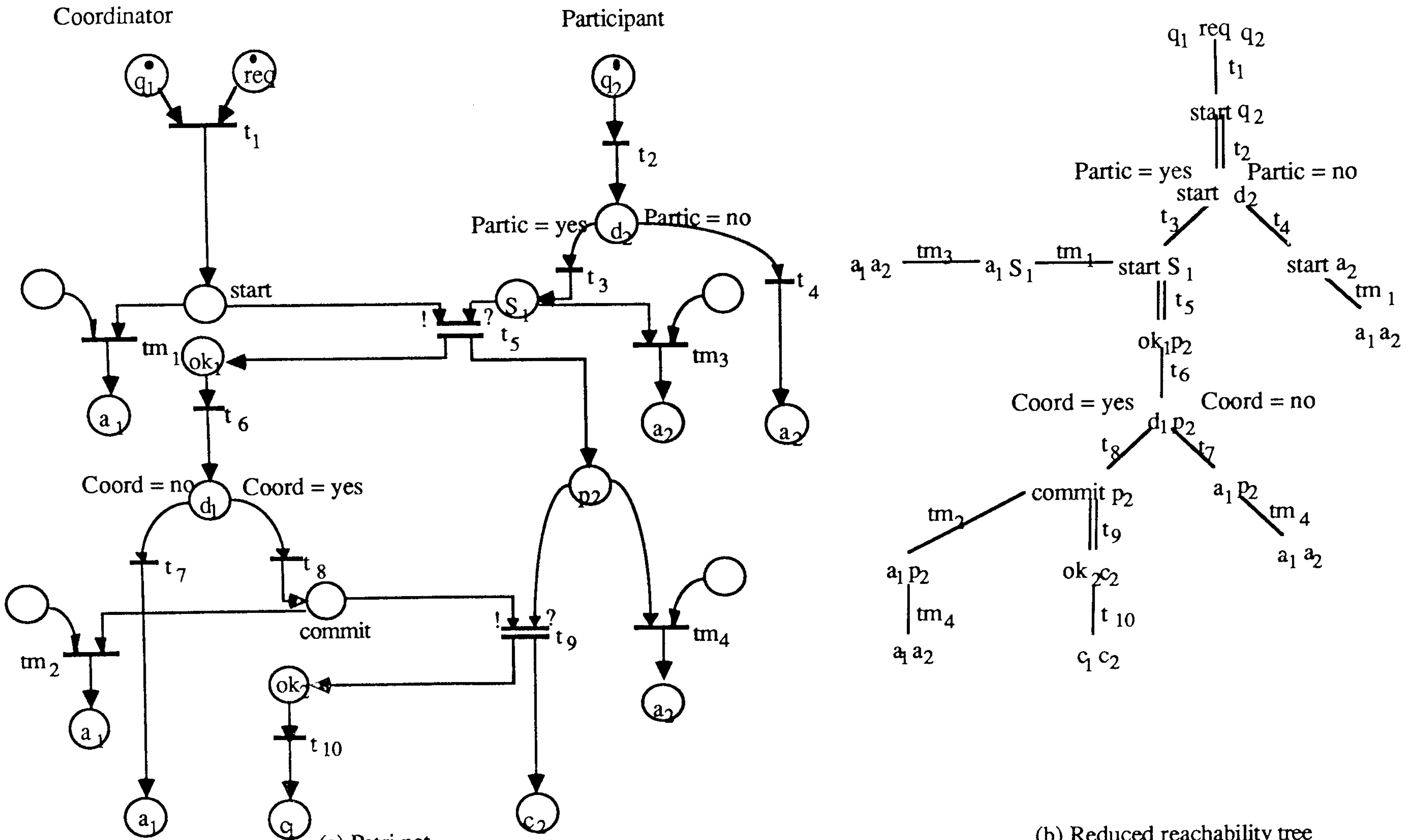
The reachability tree shows that a consistent decision is always reached even in the event of a link failure. Each communication is of the form shown in figure (4.10b) therefore each ? and ! can be replaced by the bounded communication procedure calls. The code fragments for each process being :-

```
Proc coordinator (To.p1, From.p1)
  SEQ
    state := q1
    OutputOrFail.t (..,start,..)
    IF
      failed
        -- link down
        state := a1
      NOT failed
        state := w1
        InputOrFail.t (..,mess,..)
        IF
          failed
            -- link down or p1 = no
            state := a1
          NOT failed
            -- p1 replied yes
            IF
              NOT coord.willing
                state := a1
              coord.willing
                state := p1
                OutputOrFail.t (..,commit,..)
                IF
                  failed
                    -- link down
                    state := a1
                  NOT failed
                    state := c1
:
```

```
Proc participant (To.coord, From.coord)
  SEQ
    state := q2
    InputOrFail.t (..,mess,..)
    IF
      failed
        -- link down
        state := a2
      NOT failed
        IF
          NOT partic.willing
            state := a2
          partic.willing
            OutputOrFail.t (..,yes,..)
            IF
              failed
                -- link down
                state := a2
              NOT failed
                state := p2
                InputOrFail.t (..,mess,..)
                IF
                  failed
                    -- link down
                    state := a2
                  NOT failed
                    state := c2
:
```

These optimizations have removed the no and abort messages which would only be sent if either site voted to abort, removing the ack message saves one message during commitment. One disadvantage of this protocol is that it can only be used when the communication links are dedicated to these control messages so that the propagation delays can be calculated accurately. If the timeouts are not accurate then false timeouts can occur when both sites could have gone on to commit. This disadvantage is outweighed by the safety aspect, it may be far better to have a false timeout than an inconsistent or late decision which may cause an unsafe event.

From the previous optimized protocol of figure (5.3) it can be seen that during commitment the ok_1 message from the participant is followed by a yes message informing the coordinator that the participant is able to commit. This appears to be redundant, it is now suggested how it may be possible to remove this message but only under certain conditions. The protocol suggested is shown in figure (5.4) :-



(a) Petri net

(b) Reduced reachability tree

Fig.(5.4) Optimal extended 2 phase commit protocol using synchronous communications

In this protocol the yes message is replaced by the ok_1 message, this has the disadvantage of removing the synchronization at the start of the protocol. These optimizations lead to a loss of desirable qualities and also the introduction of undesirable constraints. For example the start of each site is now assumed to be synchronous within a known time limit. That is, if the start time of the participant is not known accurately then the timeout period tm_1 cannot be calculated. It is also assumed that each participant knows what actions to perform. This is because the start message would normally include the work each participant should do but now the participant has to decide locally before the start message is received.

The participant now checks to see if its actions can be performed, if they cannot then it aborts immediately but if they are possible then it waits for a start message from the coordinator. Once this is received it enters a prepared state and waits for a commit command or timeouts to an abort state. The coordinator after receiving the ok_1 message then checks to see if its actions are possible (and checks other sites) and sends a commit message or aborts. This protocol again removes messages by assuming that a lack of a message is equivalent to a negative reply.

This protocol can not be directly extended to include more than two sites because if the decision to commit is reached and a commit message is sent to one participant but fails to be sent to another then one participant commits but the other timeouts and aborts. However from Skeen's theorem it should be possible to design a multi-site protocol which is resilient to a single link failure because providing consistency between connected sites is relatively easy.

To solve this problem another phase can be added to the protocol as shown in figure (5.5). When a site receives a commit message it now enters a 'prepared to commit' (R_1 or R_2) state in which it either timeouts and commits or receives an abort message. In the event of a timeout it is reasonable to assume a commit because we have assumed only one link failure will occur and so the last abort messages will always be received. The abort messages are sent only after a different link has failed previously and are assumed safe. If links were allowed to fail during this phase then the protocol would have to be blocking and of little use in a real-time situation.

False timeouts in the protocol happen when the timeout values are set too low or a process is slower than expected. With a dedicated system false timeouts should be rare events if the values are calculated with care. False timeouts are undesirable but acceptable if safety is preserved. Another problem is the omission of the synchronization, traditional commit protocols use the start message to synchronize autonomous sites, this has been removed for optimization but could easily be included. In a practical system the use of such synchronization is probably necessary, this would involve an extra 2(N-1) messages where N is the number of sites.

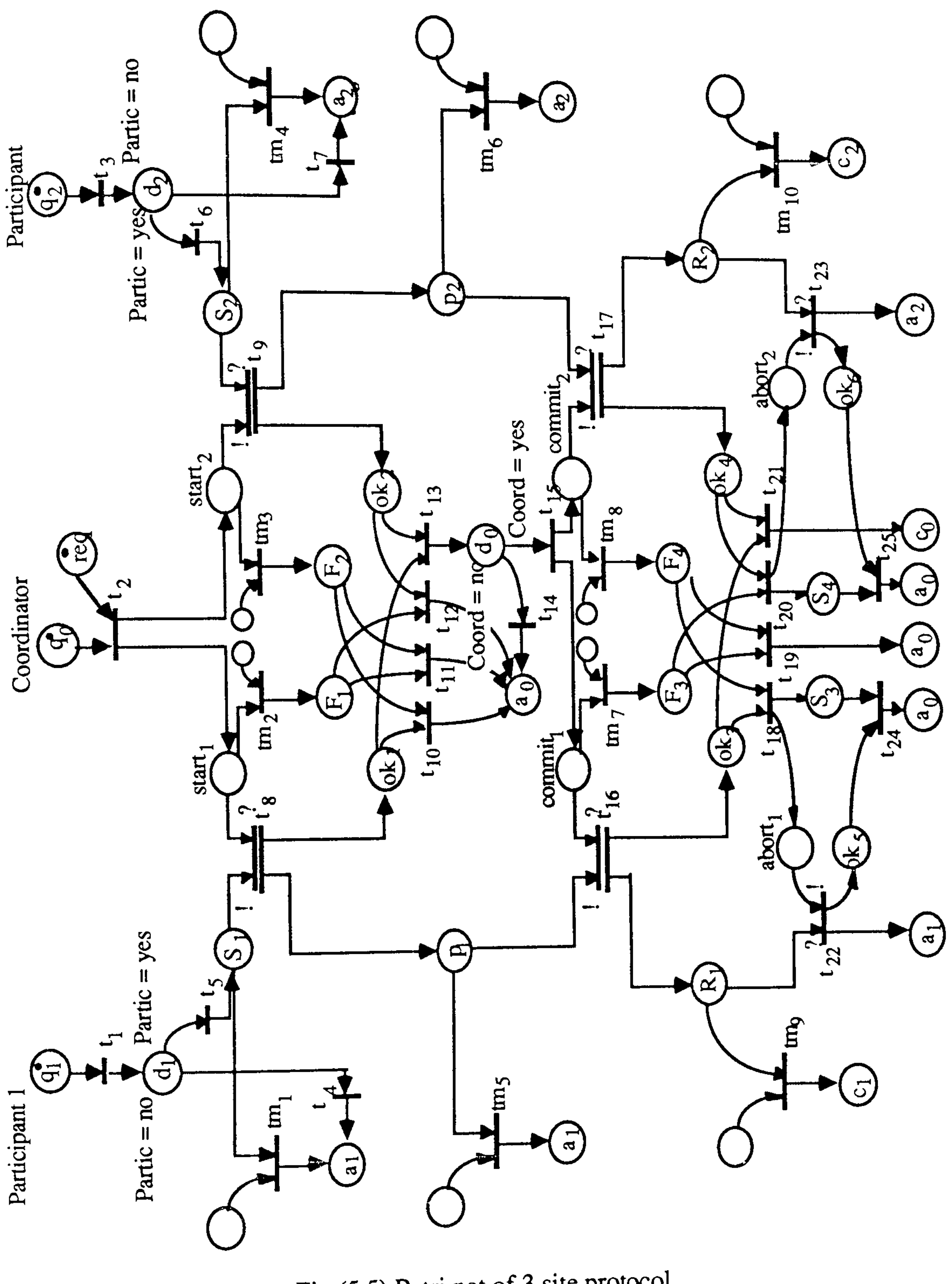


Fig.(5.5) Petri net of 3 site protocol

When the protocol of figure (5.5) is used it needs 4(N-1) messages to commit which is the same as the 3 phase commit protocol, if synchronization at the start is used a total of 6(N-1) messages are needed. This means that the message overhead for commitment is greater but since lack of a message is used to mean a negative answer the number of messages to abort is less. In this case the number of messages to abort has a lower bound of 2N-3 (if a participant says no) and an upper bound of 3(2N-3) (when a commit message fails).

The disadvantage of false timeouts and extra messages are far outweighed by the fact that consistency can be maintained even with a link failure. Since each site can recover independently from a link failure and still continue processing, any applications being controlled can still continue and real-time constraints can still be met. It would be advantageous to use such protocols in safety critical real-time control systems where timed consistency has priority over other aspects.

5.5. Discussion

To be useful a specification must be easily translated into an implementation, in this case a suitable programming language. This chapter has demonstrated how the Petri net protocol specifications can be converted almost directly into Occam programs without too many abnormalities. Special cases due to practical limitations (the number of Transputer links) have been pointed out and shown to be transformable into a communication followed by a selection.

It has also been shown that Occam is a valuable tool for programming distributed systems, possessing the necessary constructs to define parallelism, inter-process communication and non-determinism. The advantages of Occam, its simplicity and mathematical fundamentals probably outweigh any disadvantages. One disadvantage of Occam is that it has insufficient semantics for highly robust distributed systems primarily by not allowing a timeout on a send command. This has been overcome by the use of pre-defined assembler routines. Another disadvantage of Occam is its fixed type of synchronous communications which may not be flexible for all systems. This thesis proposes that this type of communications is more predictable and should be used for safety critical systems.

The use of synchronous communications for the protocols has been analysed and an optimized two site protocol has been developed. This essentially uses the the idea of implicitly allowing the acknowledgement signals to carry information. The presence of an acknowledgement is assumed to represent a positive reply whilst the lack of acknowledgement means a negative response. This uses timeouts to determine a decision and so puts greater emphasis on accurate timeout values. Also for the totally optimized case the sites must have a totally synchronized start, this would probably reduce the number of applications of such a protocol. A three site protocol using these optimizations has also been developed.

The 2 phase commit and extended 2 phase commit protocols have been implemented using Occam on a network of Transputers. The listing of these protocols with timeouts defined for link failures are given in appendix B. The implementation confirm that the 2 phase commit using synchronous communication is still a blocking protocol. They also confirm that the extended 2 phase commit can independently recover from a single link failure. The optimized extended 2 phase commit protocol has also been implemented, its listing also appears in appendix B. The results of this show that it still exhibits the same resiliency to link failures as the extended 2 phase commit but the assumed synchronization does pose problems. The calculation of timeout periods is also a greater problem in this protocol because false decisions can be made.

The following chapter will demonstrate the use of the optimized protocols in real-time applications. It will be shown how making a decision within a deadline is important in safety critical situations and why tolerating a link failure is more important than tolerating a site failure.

Chapter 6

Application of commit protocols to real-time systems

6.1. Introduction

This chapter illustrates the use of commit protocols using synchronous communications in real-time systems. Section (6.2) examines the requirements of various classes of real-time systems. Section (6.3) provides two real-time database examples which show how site and link failures can be tolerated by using the protocols developed in chapter 4. Two real-time control examples are provided in section (6.4) both of which show how important atomicity is in a control environment. The first example, a level crossing, is used to show how atomicity can be applied without any timing constraints. Timing constraints are included in the second example which shows how the protocols of chapter 4 can be applied to safety critical situations.

6.2. Characteristics of real-time systems

In general a real-time system must be predictable both functionally and temporally [Stankovic 88a]. In such a system, the basic component of work is the task which can be either periodic or non periodic. Periodic tasks are predictable and are performed once every T seconds and can be therefore scheduled with known start times. Non periodic tasks are known events which may happen at any time, therefore their start times are unknown. A processor may be executing a number of periodic tasks when a non periodic task is initiated, the processor must then be able to determine which task has higher priority. For a task to be completed in real-time, certain temporal constraints must be satisfied. A task can be characterized by the following constraints [Ramamrithran 84] :

(1) start time

(2) computation time

(3) deadline for completion

All three can be represented as absolute or relative times. For a task to be possible the following must hold:-

(deadline - start time) >= computation time

If this is not true then it is impossible for the task to be completed in time. The difference between the task completing and the deadline is known as the laxity of the task and is a measure of how much spare time there is. This laxity is very useful when attempting to schedule a number of tasks on one processor. It may be possible to use this spare time of a processor to execute another process and reduce the number of processors required.

Temporal scopes [Lee 85] are a language construct which associates timing constraints with a collection of statements. They assume that an underlying scheduler exists which uses the timing information provided. The timing constraints are provided by the user and are more explicit than simple delays. The possible attributes to a temporal scope are :

(1) Deadline for temporal scope
(2) Minimum delay before starting temporal scope
(3) Maximum delay before starting temporal scope
(4) Maximum execution time of the statements in temporal scope
(5) Maximum elapsed time of temporal scope

The maximum elapsed time is the actual time that the processor is used for the temporal scope, this includes all delays where the processor is not executing any command. An example of such a delay is when waiting for a communication. A scheduler uses the information about the elapsed time to try and schedule other tasks in the delays. One problem with this is that the delays must be defined by the user which is a very difficult task. These extra constraints are not needed for systems where the scheduler does not use user defined timing information. A Transputer based system is an example of where such timing information is irrelevant because the scheduler is micro-coded and priority based and does not use information provided by the user.

Other deadline semantics that have been included in a real-time operating system [Gheith 89] are recoverable deadlines and weak deadlines. Recoverable deadlines are useful for tolerating failures and still providing a safe system. For example if a hard real-time task is scheduled and misses its deadline because of a failure it is possible to recover by defining an exception handler which is executed with highest priority. Weak deadlines are such that when deadlines are missed partial values already generated may be acceptable by the system. This may be satisfactory for a single process system but for a distributed system it is difficult to maintain consistency between sites and is therefore rarely used.

Tasks also have different importance and when scheduling it is useful to know these so that the more important tasks take precedence. Real-time tasks can be classified as either hard real-time or soft real-time [Burns 90]. A hard real-time task is such that if its deadlines are missed then the system fails and a catastrophe may occur, for example an aircraft controller. If a task misses its deadlines but the system still functions correctly (and safely) then the task is said to be soft. In a real-time system a hard real-time task has a higher priority than a soft real-time task.

Most real-time systems will not have enough resources for each required task and as such multi-tasking must take place. A typical situation being a number of tasks which are executed on one processor. So that tasks can be run with different priorities a scheduler must be provided by the operating system, such a scheduler ensures fairness between equal priority tasks and also includes a resource manager which allocates in advance time slots for access to shared resources by specific tasks. Scheduling involves executing all hard tasks within their deadlines and also as many soft tasks as possible. This may involve pre-empting soft tasks to ensure a hard task finishes in time.

In a distributed real-time system with limited resources it may be possible to schedule and execute tasks on other nodes. Such a scheme has been presented [Ramamrithran 84] which requires each node to have additional processes which take care of scheduling tasks, bidding for tasks and dispatching tasks. When a task arrives at a node an attempt is made to schedule it, if unsuccessful the task is sent to another node. This continues until the task is either guaranteed or the deadline is missed. For this algorithm to be successful each node must be able to handle concurrency and must be connected to the other nodes by a reliable and efficient communications system.

With limited resources and sporadic tasks, scheduling is generally performed by an operating system with a kernel providing a limited set of primitives. Current operating systems provide scheduling based on priority rather than time and have limited ability to manage time and time bounded communications. Lee et al [Lee 89a] have developed a real-time kernel for use with a robot arm to provide predictable timing behaviour which uses temporal scopes to instruct the scheduling algorithm. The scheduler is priority based and classifies tasks as either imperative, hard or soft. Imperative tasks have the highest priority and are executed on a first come first serve basis whilst hard and soft tasks are executed depending upon their timing constraints. Once a hard task is scheduled its deadlines are guaranteed but soft tasks may not complete in time. If a hard task violates a timing constraint because of a failure, the tasks priority is raised to that of imperative. Its exception handler is then executed immediately. This allows tasks to start off as non real-time and as soon as a timing constraint is specified it becomes a real-time task. Timing constraints are specified by temporal scopes extended with a flag which indicates soft or hard constraints. The advantage of this system is that using temporal scopes means that the scheduler can use timing information not just priority to derive its schedule. An extension to this work would be to allow tasks to be scheduled on nodes other than where they were initiated.

Real-time systems are typically used in control applications with sensors and actuators interfacing to the external environment. Such control systems are using ever increasing amounts of information and are now being equipped with database systems for efficient storage and manipulation. In database systems the unit of work is the transaction. If the database has to respond within predefined time limits the system is known as a real-time database system. Transactions are being extended with timing constraints [Stankovic 88b, Abbot 88]. However problems still remain when all the properties of a transactions are required. Properties such as concurrency control and atomicity cause problems in predicting computation times because recovery times and data locking times can be unpredictable. Usual operating features such as paging, buffering and resource allocation also make deadline prediction more difficult if not impossible.

Many transactions in a real-time database system have deadlines which are not critical and so a more flexible model for time constraints is being developed [Abbot 88]. The method assigns a function which varies with time to each transaction, the sign and magnitude of the function expresses how useful the result of the transaction is. For example if a hard real-time transaction completes before the deadline then the function is a high positive number but once the deadline is passed the function becomes zero. Although flexible the problem still remains of how and when to classify the transactions, also what happens if the transactions are classified incorrectly? One short coming is that the method does not include concurrency control in its outline. Concurrency control is important because to achieve serializability certain tasks may need to be rescheduled which may violate the deadlines.

One problem of real-time database systems which needs further investigation is that of recovering from failures in a timely fashion. Most work in this area concentrates on main memory databases which are not distributed [Singhal 88]. When a site fails which is involved in a transaction with a deadline, it must recover and return the required results before the deadline expires. In practice the recovery time of the site is unpredictable and the analysis intractable. A similar situation also arises during commitment of a transaction. Consider a transaction using a blocking commit protocol such as the 2 phase commit, if a site fails the transaction has to wait until the failed site recovers before terminating. This would involve keeping data locked and possibly unavailable to other transactions. Other transactions may therefore miss their deadlines. Section (6.3) proposes the use of non–blocking protocols to solve this problem. With a non-blocking protocol, if a site fails, the operational sites can terminate the transaction thus releasing locks. Upon recovery the failed site recovers to a consistent state. This means although the original transaction cannot keep its deadline it will release locks on data in time for other transactions to meet their deadlines. Also if a single link between two sites fails it is possible to continue processing at both sites independently if the network is such that no messages are lost. A solution is proposed in section (6.3) which uses the protocols developed in chapter 4.

Other properties required in a real-time system are robustness and flexibility. Robustness is a combination of reliability and availability and should provide an acceptable service in the presence of failures. An example of a system where robustness is very important is in an avionics control system where system failure is expected to be extremely rare. In such a system if a processor fails then for the system to be robust control must be maintained. This can be achieved by having another processor execute the task of the failed processor. It is difficult to tolerate site failures in real-time because site recovery is unpredictable and additional hardware must be used and the switching of tasks is required. Section (6.4) illustrates how a site failure can be tolerated without additional hardware using a commit protocol but not within a deadline.

Consideration is also given to link failures in such systems. These leave the processors able to function correctly but unable to communicate. Redundant links may be used to tolerate a link failure but for a system with limited resources this is impractical. Since the processors are still functioning, theoretically it is possible to still keep control of the system. Since coordination is impossible the only way is for all (now) independent sites to assume a state known to be safe. This is illustrated in section (6.4) with an example which requires the safe state to be achieved within a deadline.

The property of flexibility can be defined in terms of static or dynamic flexibility. Static flexibility is based upon modularity where functions are identified with modules and if the functions change (eg. changing the product shape) then it is only necessary to change a few modules and not the majority of the system. Dynamic flexibility ensures that the system performance is maximized as the working conditions change. The system must be able to change instantaneously at unpredictable times if working conditions change (eg. loads or faults occurring). Both static and dynamic flexibility must be easily implemented if the system requirements or conditions are expected to change frequently. The examples in the following sections are static and therefore their functions are assumed constant. They are designed to tolerate faults thus demonstrating robustness. However the methods illustrated are applicable to flexible systems if the communication network remains fixed. To achieve full flexibility all sites must be able to coordinate with any other and so each site must be able to be designated a coordinator or a participant. The role of each site would also have to be switchable at any time. This increases the complexity and is not considered in the examples.

In the following applications reference is made to the Transputer [Inmos 88] which is a VLSI chip with built in processor, memory and four point-to-point synchronous communications modules which can execute in parallel with the processor. It has been used in real-time control systems[Leppalla 87, Flemming 87, Irwin 90, Chull 89] because of its high processing power, low cost, modularity and simplicity of interconnection.

Interconnections are via communication links which provide bi-directional synchronous communications and can operate simultaneously thus providing a very high message throughput. Being synchronized means that communication is very reliable because messages cannot be lost or received out of order as in asynchronous communications.

The Transputer allows concurrency by using a microcoded scheduler based upon time slicing. Priority allocation to processes can be performed directly from Occam but this allows only two priorities. Another problem is that processes are scheduled on priority and not timing constraints. Although not ideal for real-time systems the microcoded scheduler does alleviate the designer from scheduling problems. Eventually as costs fall the Transputer will be used in systems where one process executes on one processor thus removing the need for a scheduler. Other advantages of the Transputer include its efficient communication system and the ability to implement Occam programs directly. This is an advantage because Occam is based on the CSP process model and follows its rules for transformations. Thus an Occam program can be transformed into another which can be proved to provide the same function [Roscoe 88]. This is usually performed in an attempt to reduce the hardware required. The transformation of real-time programs is much more difficult because time constraints must also be met [Moitra 90] this involves taking into account the scheduling strategy used.

6.3. Commit protocols in real-time database systems

This section examines the use of commit protocols in real-time database systems. It proposes the use of non-blocking protocols to provide transactions which can satisfy timing constraints. It also proposes the use of synchronous communications for commit protocols where the application is safety critical and link failures are expected. Two examples are used to demonstrate the proposals.

An example of a real-time database system can be found in a radar system tracking an aircraft, the position, direction and speed being stored in the database. Such systems are often distributed and comprise multiple sensors, processors and data. If a weapon system uses this information then the storage and retrieval time of the information must be predictable so that the system knows it is trailing by a known amount. This delay can them be taken into account in the design of the weapon and guidance system. This causes problems because database systems generally have slow response times for data access because of disk i/o and the need to maintain consistency. Data consistency is often achieved by serializing transactions but which can cause unpredictable delays because transactions may be restarted or blocked. The previous performance problems are solved by either using main memory databases or trading a feature such as serializability [Singhal 88].

Main memory databases can be used to improve the performance of a traditional database. This is because they possess a large fast main memory which is used to store part if not all the database thus reducing database access time. The main problem with this technique is the high cost involved but the cost of memory may continue to fall in the future. Another problem is that of crash recovery by using stable storage. This type of storage is slow and a typical recovery technique such as reading a log record may not be tolerable for real-time applications. The time between a site failing and the recovery process starting is also unpredictable.

Performance of a database system can also be increased by having less stringent transactions [Stankovic 88b]. One method is to sacrifice the serializability property of transactions. The correctness of a database is actually decided by predefined integrity constraints on the data. A simple example of an integrity constraint is that a bank account must contain an amount greater than zero (or the overdraft limit). The number of these constraints is typically large and so serializability is used instead [Davidson 85]. Serializability is used in database systems to ensure that concurrent execution of transactions has the same effect as if they were executed serially so as to maintain a correct database [Bernstein 87a]. Since serializability is achieved by blocking or restarting transactions the time to complete a transaction in a concurrent environment may be greater than if executed in isolation. Real-time database systems usually have a fixed number of transactions to perform and it is possible to specify a small set of integrity constraints which can be used instead of serializability. This technique has been used to design a quorum protocol [Lin 88] which allows greater data availability in the presence of a network partitioning.

Real-time databases are expected to show resiliency to failures as well as predictable response times. Resiliency to failures can be provided by a distributed database with replicated data. Distribution has the advantage that if one site crashes the others can still continue processing. Replication of the data enhances data availability in the event of a failure and can also be used to improve transaction response time by judicious placement of the copies.

Full replication of the database means that a transaction can still complete even if any site fails. This is wasteful of resources and is not common practice, usually only part of the database is replicated and rarely at every site. To increase performance copies of data can be placed where they are expected to be accessed frequently thus removing long communication delays. A problem with replicating data is ensuring that all copies act as one, even in the presence of failures. For example if a transaction updates a data item which has a copy unavailable because of a site failure, upon recovering the failed site must update its copy so as not to be inconsistent.

A number of algorithms exist to control multiple copies of data [Bhargava 87, Son 87]. Most require that each site knows the status of every other site, either operational, non-operational or recovering. Another concept common to both is the read-one / write-all-available paradigm where a read transaction acquires a lock on one copy whereas a write transaction must be able to lock all available copies. Fail locks [Bhargava 87] are used to inform operational sites that data being updated has copies at failed sites. Keeping tables of fail locks at each site is a disadvantage because of the communication overheads incurred. Son [Son 87] exploits the dominance of read-only transactions to enhance performance, one problem being that the read-only transactions must be identified before execution. Such resiliency to failures helps to achieve real-time response but the control of the copies has an adverse effect.

When a transaction has to complete within a deadline two conflicting situations arise, database consistency and data availability. If database consistency is preserved it may mean keeping locks on data items thus reducing data availability. If database inconsistency can be tolerated it may be better to remove locks on data so that other transactions can complete within their deadlines. A real-time database has two types of consistency to preserve, internal and external. Internal consistency of the database is provided by integrity constraints. External consistency means the database must model the outside world accurately which implies data has a limited life span so the database must be updated frequently. Out of date data is sometimes useful in real-time databases where a transaction response time is critical but the value is not.

The main problems associated with real-time databases are providing concurrency control of transactions within deadlines and also recovery from site and link failures. Concurrency control algorithms must be optimized so that blocking of transactions is minimal. A high performance concurrency control algorithm has been developed [Singhal 88] which removes communication delays from the blocking time of a transaction. Recovery techniques must also be developed which allow fast recovery from site failures and continued processing of transactions when a link fails.

6.3.1. Example 1 : site failures

This example demonstrates how non-blocking protocols can be used in real-time database systems to allow transactions to satisfy deadlines when sites fail. Consider the database system shown in figure (6.1) where the transaction T_u is initiated at db_1 and expects all three databases to be updated.

Assume that a copy of a data item X resides in each of the databases, let the copies be denoted x_1, x_2, x_3. Also assume that the transaction T_u is to update X and once updated X is then read by another transaction. For the second transaction to satisfy its time constraints T_u is specified to complete within the deadline D.

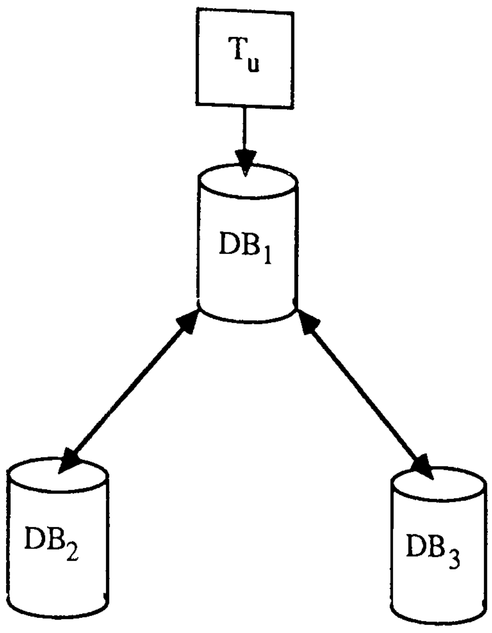


Fig.(6.1) Example database system

Now consider the situation where T_u is about to commit but db_3 fails. If a blocking commit protocol were used T_u cannot complete until the failure is repaired, this may be sometime after D and so the deadline is missed. To solve this a non-blocking protocol extended with timeouts can be used. T_u detects that db_3 has failed and terminates appropriately at db_1 and db_2, a result is then available at D for the next transaction. Upon recovering db_3 examines its log record and brings itself to a state consistent with db_1 and db_2.

6.3.2. Example 2 : link failures

This example demonstrates how a commit protocol using synchronous communications can provide consistency even if a link fails. It also shows how the timing constraints of a transaction can still be satisfied.

Consider again the database system of figure (6.1) and assume that Tu is a transaction that updates share prices and the three databases are each local to a different stockbroker's office. If the major cause of failure is expected to be a communication link failure then the system must be designed to ensure that no stockbroker receives information that the others do not.

If a non-blocking protocol with asynchronous communications is used it is possible for the databases to become inconsistent because messages can be lost. As an example consider T_u sending commit messages to db_1, db_2 and db_3. It is possible for the messages to be received by db_1 and db_2 but for the commit message to db_3 to become lost, the share price at db_3 will not then be updated. If instead a commit protocol using synchronous communications such as described in chapter 4 was used messages could not be lost and this inconsistency would not occur. The protocol was shown to be correct in the presence of a link failure in chapter 4.

The transaction Tu could be periodic so that the share price is updated every P minutes. A deadline D can now be placed on the completion of Tu so that $D < P$. The timeout values in the protocol can be assigned so that the transaction provides a decision by D at the latest. Any timeout that is not satisfied can be assumed to imply a link failure. The action taken to maintain consistency is decided by the protocol and has been explained in chapter 4.

6.3.3. Summary

Both examples show that non-blocking protocols are useful in a real-time database to allow transactions to meet their deadlines. Example 1 demonstrated how a replicated database could still satisfy its timing constraints even if a site crashed, the increased data availability was due to the non-blocking commit protocol. If consistency is considered more important than availability then a non-blocking commit protocol with synchronous communications can be used, such a situation occurs in safety critical systems. The communications used in example 2 imply that a link failure is detected quickly and easily.

A disadvantage of using a non-blocking protocol is that if the cause of failure (site / link) cannot be determined then an inconsistent result can occur. Disadvantages of using a protocol with synchronous point-to-point communications as in example 2 are mainly due to its inherent inflexibility. The network must be defined beforehand and cannot be changed dynamically. This would appear acceptable for most real-time databases which are usually static but a general purpose database system usually needs more flexibility.

6.4. Applications in real-time control systems

This section uses two illustrative examples to demonstrate the use of atomic commit protocols using synchronous communications. Atomicity is shown to be a necessary property in both applications with timed atomicity required in the second.

It is shown how the use of synchronous communications provides extra resiliency to network partitioning than can be expected by using asynchronous communications. The examples are both plausible for Transputer implementation.

6.4.1. Level crossing example

As a simple example of a control system consider the level crossing outlined in figure (6.2). The gate G_1 can be in one of the states {UP, DOWN} whilst the state of the lights L_1 and L_2 belong to the set {STOP, GO}. A token in place P_1 represents the train lights in the state STOP, whilst a token in place P_2 represents the state GO, similarly for the car lights and places P_3 and P_4.

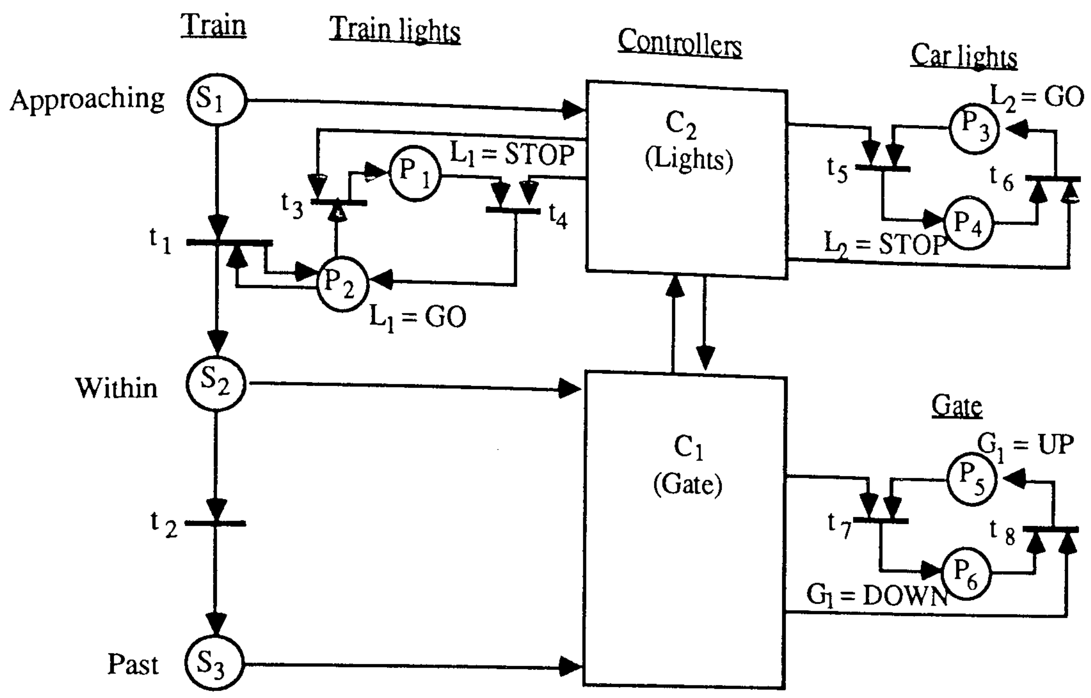


Fig.(6.2) Outline Petri net of level crossing example

Figure (6.2) shows that the position of the train is detected by three sensors S_1, S_2 and S_3, S_1 detects when a train is approaching the crossing, S_2 indicates when a train is within the crossing whilst S_3 signals when the train is clear of the crossing. A token in one of these places represents the position of a train. If more than one token is allowed to be in the union of S1, S2 and S3 then this models the presence of a number of trains. The gate is raised and lowered by a controller C_1, which takes its inputs from S_2 and S_3, another controller, C_2 uses the lights L_1 and L_2 to control the position of trains and cars respectively. The only communication between the controllers is by message passing which can not be assumed perfect. The coordination between the lights and the gate must now be performed so that a hazardous situation cannot happen, the situation to be avoided in this case is when L_1 = GO, L_2 = GO and G_1 = UP. Another not so serious case to be avoided is L_1 = STOP, L_2 = STOP and G_1 = DOWN which is not dangerous just unintended.

The action of the controllers is twofold, they must detect the position of a train and control the gate and lights accordingly, secondly they must also ensure that the afore mentioned hazardous states do not occur. The gate should be lowered when a train approaches the crossing and cannot be stopped in time, it should then be raised once the train has left the crossing. Another problem occurs when more than one train is allowed at the same time, e.g. tokens in S_1 and S_3.

To show the usefulness of commit protocols in such an application, part of the control of the gate is now replaced with such a protocol. For clarity only the action of raising the gate is shown, the reverse, the lowering of the gate when a train approaches is not explained but would follow in a similar manner. First of all the action of the controller is explained for one train and then when more than one train is allowed so as to emphasize the usefulness of the commit protocol.

Assume that initially G_1 = DOWN, L_1 = Go and L_2 = STOP and a train is within the crossing (token in S_2). When the train is clear of the crossing (a token in S3), the gate controller senses this and initiates a commit protocol between C1 and C2 with the aim of raising the gate and changing the lights. This property can be provided by an atomic commit protocol as shown below in figure (6.3) with C_1 as the coordinator process and C_2 as the participant.

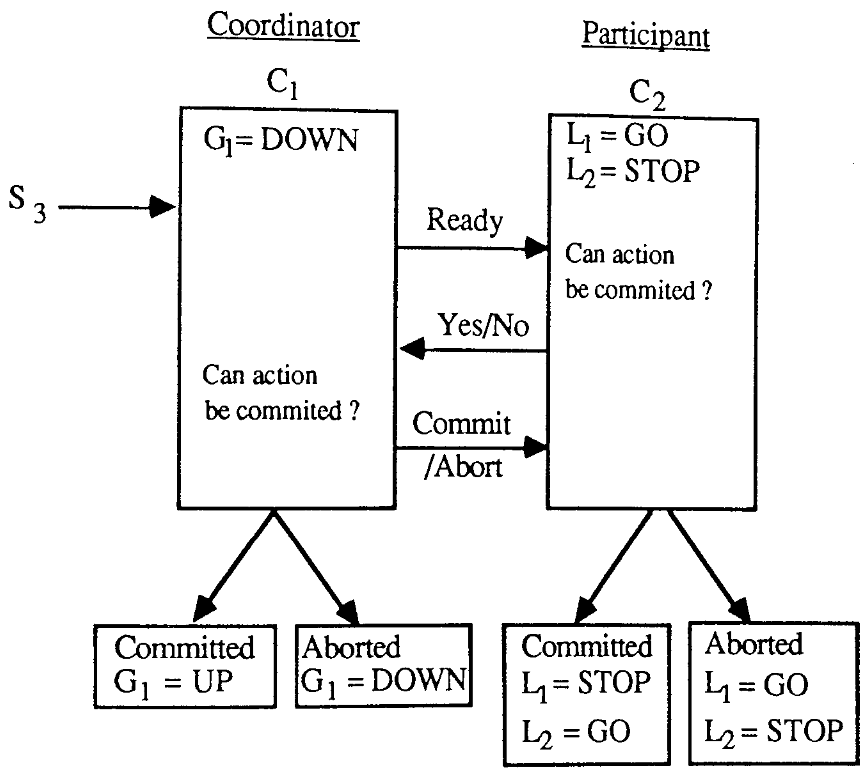


Fig.(6.3) Level crossing decision

If failures do not occur then the final state of the system will be either {G_1 = UP, L_1 = STOP, L_2 = GO} or the same as the initial state, thus preserving the safety of the system. The yes/no decision made by the participant is based upon whether another train is approaching (no decision) or not (yes decision), this information is provided by the state of S_1. The gate controller C_1 has the final decision and checks to ensure the gate can be raised, for example, ensuring that there is not a train within the crossing or the gate is not already up. Without failures the above algorithm is correct but is not resilient to messages being lost. For example if the commit message to C_2 is lost the gate will be raised but the lights will not change thus causing a hazard.

To prevent this situation the robust extended 2 phase commit protocol using synchronous communications (see chapter 4) can be used. Initially only link failures are allowed, processors are assumed to always function correctly. This is probably not realistic but the probability of a communication link failing is probably greater than a processor failing in such an environment. To show the use of commit protocols in this application the controllers can now be replaced by the Petri net model of such a protocol, in this case figure (5.3) is used. For clarity, only the Petri net model of the actions involved in raising the gate are shown in figure (6.4). An attempt to raise the gate is only made when a train has left the crossing, the gate should be lifted if and only if it is safe to do so. The reduced reachability tree of figure (6.5) is derived from figure (6.4) with an initial state of (S_3, q_1, q_2, L_1, L_3, G_1). It shows that if only one train is allowed the actions performed by the controllers are consistent and safe, even in the event of a link failure. Also link failures are modelled and the protocol is shown to preserve a safe state for all link failures. Failure transitions are shown by a double bar and modelled as not firing (equivalent to a link being disconnected).

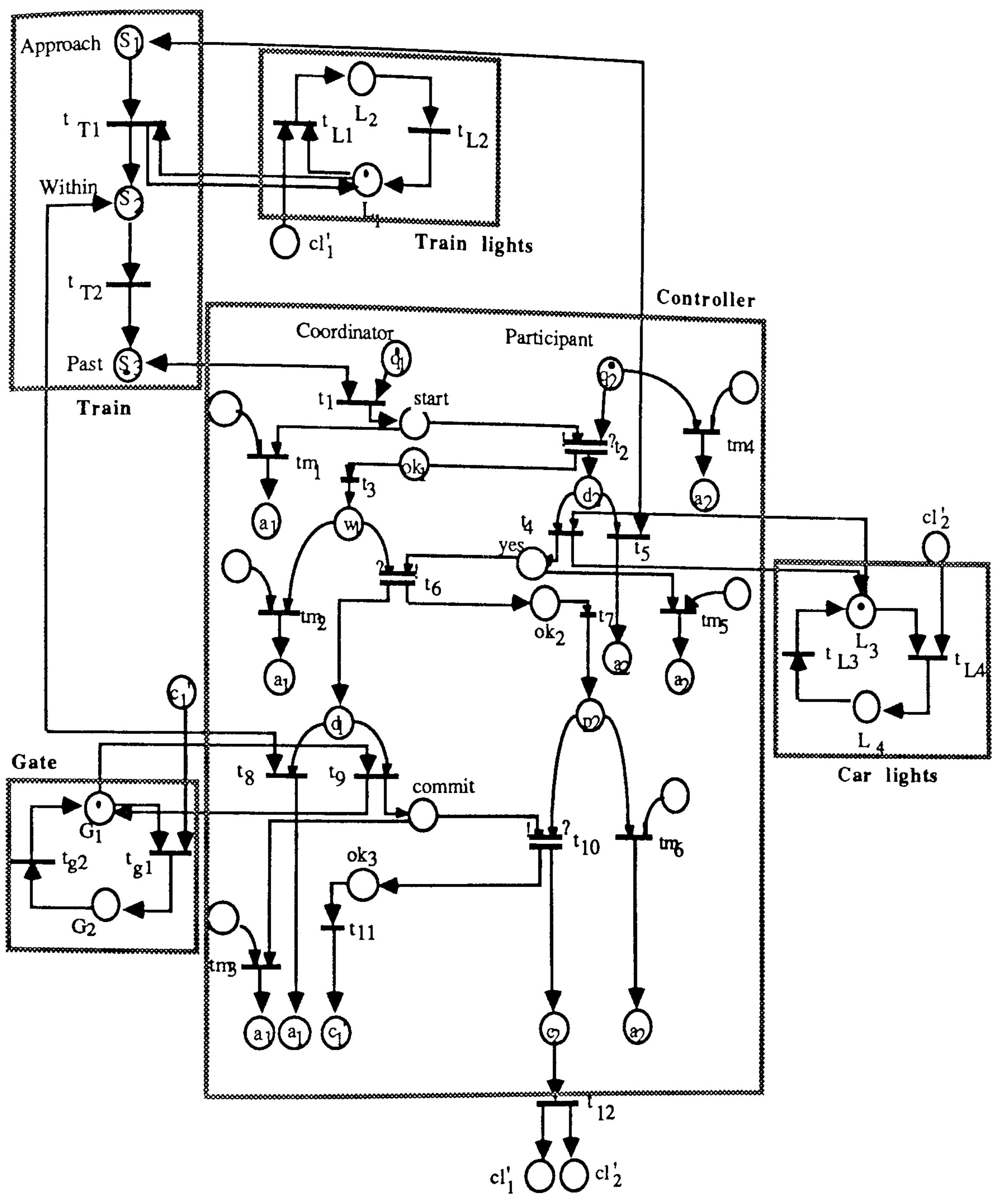


Fig.(6.4) Petri net of raising gate with possible link failures

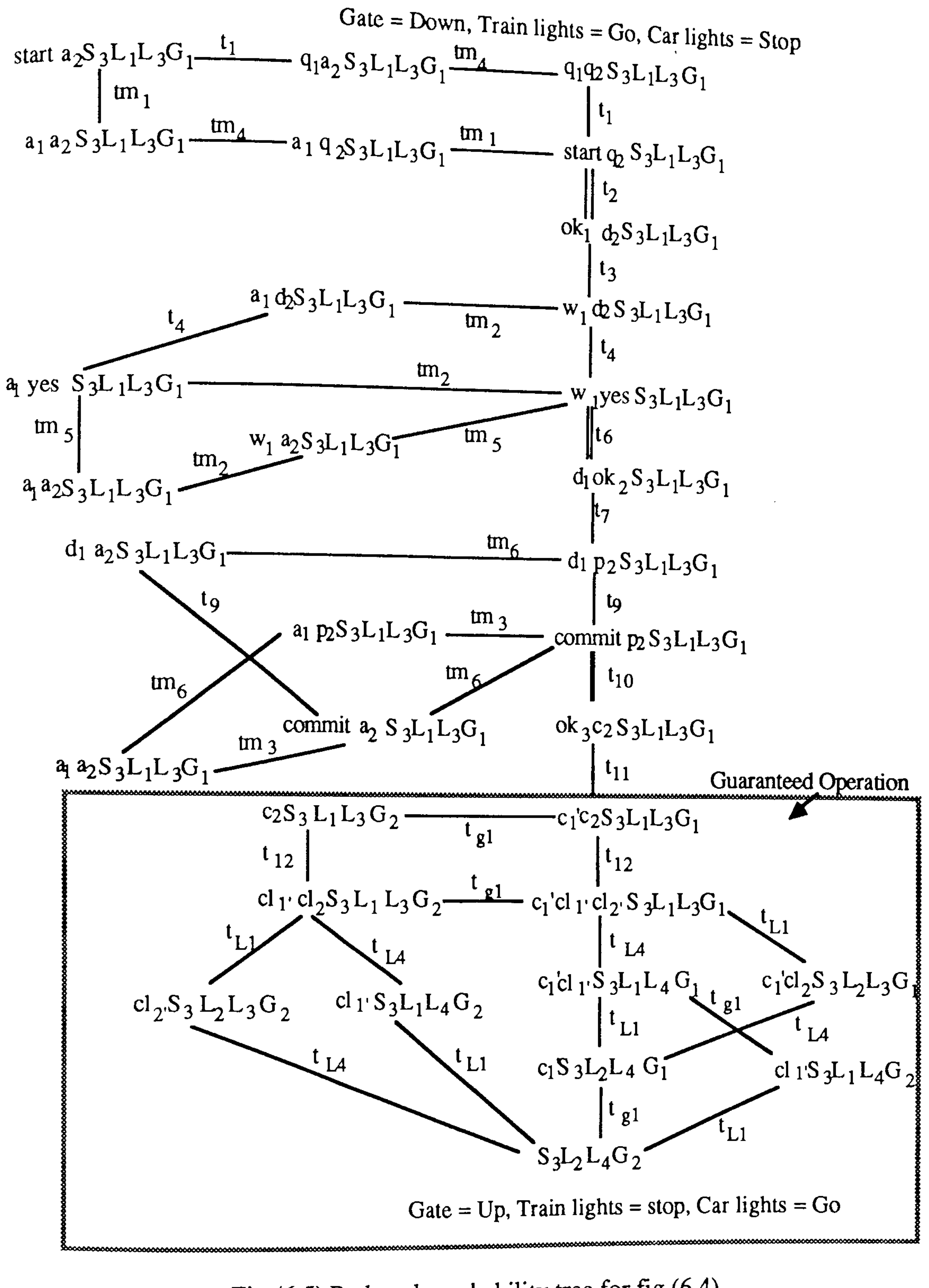


Fig.(6.5) Reduced reachability tree for fig.(6.4)

Analysis of the reachability tree shows that if a link failure does occur the undesired state $\{G_1 = \text{UP}, L_1 = \text{GO}, L_2 = \text{STOP/GO}\}$ is never reached. If asynchronous communications were used instead when a link fails messages may be lost and so this property cannot be guaranteed. Once the transition t_{11} has fired the operation of the system can be guaranteed (no site failures are allowed) and so the final state will eventually be $\{G_1 = \text{UP}, L_1 = \text{STOP}, L_2 = \text{GO}\}$.

To emphasize the usefulness of such a protocol the model can now be extended so that more than one train is allowed.

This can be modelled in the Petri net of figure (6.4) by allowing more than one token at the same time in the places S_1, S_2 and S_3. For example, a token in S_1 and S_3 represents the state when a train is on the approach to the crossing as one has just left. To demonstrate this situation it is assumed that a token may appear in S_1 at any time before the final decision is made. The train represented by this token may enter the level crossing before the train lights change to STOP or it may be prevented from entering by the lights changing in time. It is assumed here that the signal from S1 to the controller indicates that the train is too close to the crossing to be stopped by the train lights. If this situation occurs then the decision of the controllers should be to abort the raising of the gate and prevent the unsafe state. It is now shown how the decision to raise the gate can be aborted if another train is approaching close to the first train.

As before the commit protocol allows a decision to be made at each participating site, the coordinator and participant in this case. The decision made by the participant is to ensure that it is safe to change the state of the light L_1 from GO to STOP. The function of the coordinator is to ensure that it is safe to raise the gate, therefore if a train is within the crossing the gate must not be raised. To demonstrate the actions when two trains are allowed consider figure (6.4), and allow another token to appear in S_1 after the first train has initiated the decision mechanism (i.e. t_1 has just fired). As soon as a token appears in place d_1, both t_4 and t_5 are enabled, that is the second train is detected by the participant. Since this train is too close to be stopped, the raising of the gate must be aborted, hence t_5 is assumed to fire. Assuming that the train does not enter the crossing (t_{T1} does not fire), then this situation is demonstrated by the partial reachability tree of figure (6.6). The train is shown appearing after t_1 has fired but could happen any time before t_5.

$$\begin{tikzcd}
q_1 q_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_1"] \\
\mathrm{start} q_2 S_1 S_3 L_1 L_3 G_1 \arrow[d, dash, shift left, "t_2"] \arrow[d, dash] \\
ok_1 d_2 S_1 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_3"] \\
w_1 d_2 S_1 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_5"] \\
w_1 a_2 S_1 S_3 L_1 L_3 G_1 \arrow[d, dash, "tm_2"] \\
a_1 a_2 S_1 S_3 L_1 L_3 G_1
\end{tikzcd}$$

Fig.(6.6) Partial reachability tree with second train at S_1

Depending upon the speed of the second train it may pass into the crossing before the participant reaches state d_2, if this is the case then t_4 will fire and the participant will have voted to commit the action. This is modelled by the transition t_{T1} firing before a token reaches place d_2, as shown by the partial reachability tree of figure (6.7). However, this does not cause a problem because the coordinator detects this situation at place d_1 and aborts the decision (i.e. t_8 fires). The abort decision is taken although a request to raise the gate is made because it is unsafe to do so since a train is within the crossing. This shows how a safe system is maintained.

$$\begin{tikzcd}
q_1 q_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_1"] \\
\mathrm{start}\, q_2 S_1 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_{T1}"] \\
\mathrm{start}\, q_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, equal, "t_2"] \\
ok_1 d_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_3"] \\
w_1 d_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_4"] \\
w_1 yes S_2 S_3 L_1 L_3 G_1 \arrow[d, equal, "t_6"] \\
d_1 ok_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_8"] \\
a_1 ok_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "t_7"] \\
a_1 p_2 S_2 S_3 L_1 L_3 G_1 \arrow[d, dash, "tm_6"] \\
a_1 a_2 S_2 S_3 L_1 L_3 G_1
\end{tikzcd}$$

Fig.(6.7) Partial reachability tree with second train at S2

Such a protocol can be used to provide resiliency to site failures but obviously timing constraints cannot be applied in such a situation. To be resilient to site failures the commit protocol has to write 'log records' to stable storage for use during recovery. One assumption that must be made is that a site cannot fail whilst sending a message or writing a log record. These are realistic assumptions because the time spent sending a message will be small and also log records can be written asynchronously or to a spooler. So as not to be too restrictive a site is allowed to fail before an acknowledgement is received (modelled as an acknowledgement place losing a token). Coordinator site failures can now be added to the system model as shown in figure (6.8), participant site failures are omitted for clarity but can be placed in the same way. Note failure transitions are added to the acknowledgement places (ok_1 - ok_3) but not the message places (start, yes, commit). The timeouts used in the participant to provide resiliency to the failures are also shown.

To provide resiliency to the site failures, when the site recovers it must terminate itself consistently with all the others (the participant in this case). This is achieved by writing a log record as the protocol progresses and upon recovery using this to decide the new state. Writing of the log records can now be assigned to the transitions as follows :-

t_1 = write 'ready' log

t_8 = write 'abort decision' log

t_9 = write 'commit decision' log

Now consider a failure of the coordinator C_1 whilst waiting to receive the acknowledgement ok_3, the participant having committed its action and changed the lights. The state of the external system is now; the gate is lowered, the train lights are on STOP and the car lights are on GO. Although the position of the gate is not correct with respect to the lights it is not hazardous and is safe. Obviously this situation cannot be changed because the coordinator is unable to do anything. Once it recovers the coordinator checks its log record and finds a 'commit decision', it then commits its action and raises the gate correcting the situation. Since the recovery time of the coordinator is unpredictable, timing constraints cannot be applied here and so the system is in an incorrect state for an indeterminable time, but is eventually corrected. The recovery process executes accordingly for failures in all other states. The reachability tree is omitted because it is almost the same as figure (6.4) but shows consistency is always achieved, even if eventually. One advantage is that if more than two sites are used the sites that do remain operational can always be made consistent and safe because the failed site will be made consistent when it recovers.

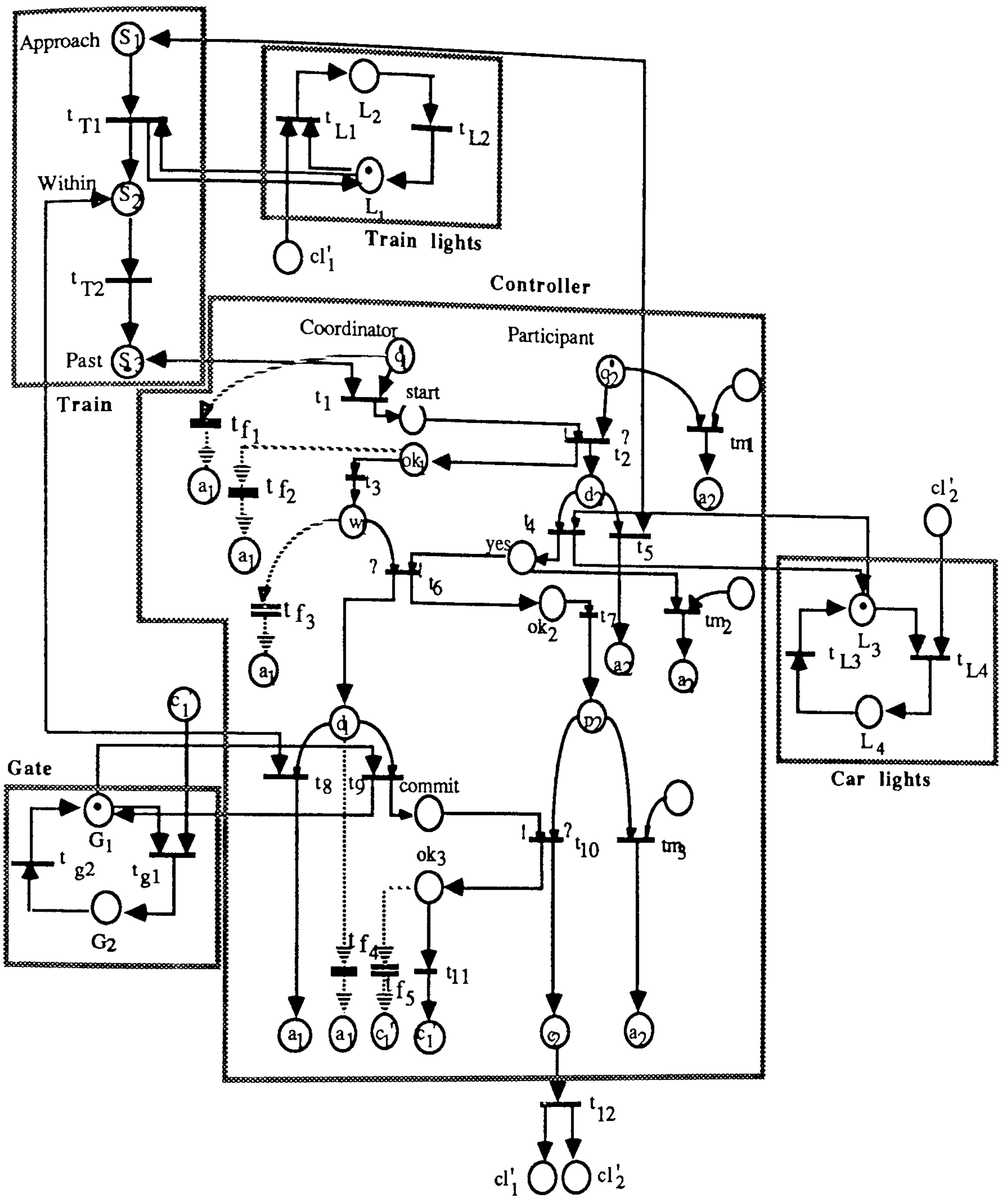


Fig.(6.8) Petri net of raising gate with possible site failures

6.4.2. Drum and Slider example

As an example of an atomic action in which timing constraints must be satisfied consider the synchronization of a drum and slider motion as outlined in figure (6.9).

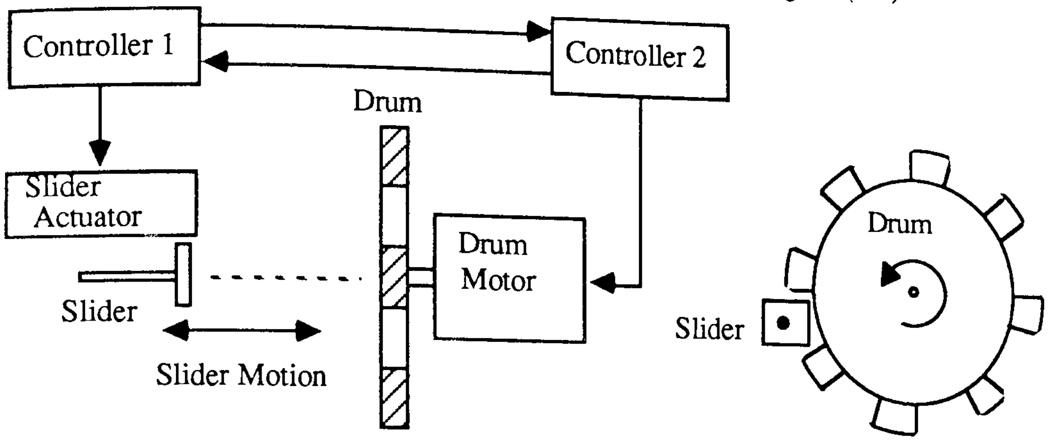


Fig.(6.9) Outline of arbor drum control system

An object is placed within one of the arbors on the periphery of the drum, the drum then rotates until the object is aligned with the slider. Once aligned the slider is inserted pushing the object from the arbor, the slider is then retracted ready for the next operation. Both of the motions are effected by motors under microprocessor control.

The synchronization of this system is critical because any slight error could cause the drum and slider to collide which is dangerous and costly to repair. For instance this could happen if the drum started to rotate with the slider still inserted in an arbor or if Controller 1 attempted to insert the slider before the drum had moved into alignment. The system must be designed so that these events never occur even if failures occur. This section proposes the use of the previously developed commit protocols to prevent such events and shows how a link failure between the two controllers can be tolerated. This section also shows how Timed Petri net models can be used during the design to ensure correct functionality and timeliness.

Timeliness is important because the slider must be able to be inserted and removed before the drum starts to rotate, similarly the drum must move from one correct position to its next before the slider reaches the drum perimeter.

One solution would be to totally synchronize the two events so that the drum only moves when the slider is stopped and out of the drum, and the slider only moves when the drum is stationary and in the correct position. This solution is safe but reduces the concurrency of the system and since the application of such a system is for high speed machinery the concurrency must be maximized wherever possible to increase performance.

A better solution is to allow the slider to continue moving towards the drum until a point is reached where it can commit (be inserted) or abort (stopped as fast as possible). This point is termed the decision point and is calculated beforehand as the point nearest to the drum such that if maximum deceleration is applied to the slider it stops just short of the drum.

The decision process can be described as an all/nothing action between the drum rotation and the slider motion with both parties taking part in the decision. It must be pointed out that this decision is not as would be expected, instead at the decision point if a commit is decided, both the slider and drum continue. It is assumed (although it may be impractical) that if a commit decision is made then the drum can rotate to its new position before the slider reaches the edge of the drum. However, if an abort decision is made then the slider and drum are stopped.

By assigning the slider controller as the coordinator process and the drum controller as the participant a commit protocol using synchronous communications, such as developed in chapter 4, can be used to implement the actions. Both controllers take part in the decision, the participant sends its vote to the coordinator and then waits for the final command. The final command is based upon the participants reply and the coordinators vote, either of which may be deemed to be abort. To vote each site must be able to check if the actions they are to perform are possible. This is done by checking the current system state for abnormalities and also possibly the availability of resources in the future. If the current system state is incorrect or the required resources will not be available then the vote must be to abort.

The drum controller, upon receiving a ready message from the coordinator has to ascertain if it is possible for it to carry out the requested operation, thus it must make some checks on the current and future system states. A possible check for the participant could be to ensure that the slider is not within the drum already and to ensure that the drum is in a safe position. Also if the controller processor is being shared by more than one process then the drum process must be able to ensure that enough resources (eg CPU) will be available for it to complete its task in time. The pre-allocation of resources is usually carried out by a resource manager, possible resources include processor cycles, memory and communication bandwidth. Once the resources have been allocated it is guaranteed that the drum will rotate to its new position by a specified time. If enough resources were not available to the participant the drum may not complete its rotation before the slider arrived at the edge. If the slider is within the drum or not enough resources can be allocated then the participant replies no otherwise it sends a yes message.

The slider controller also has to decide if its actions are possible in a timely manner. The slider must not be inserted before the drum has completed moving into an index position. Since the time for the drum to rotate to its new position is guaranteed by the participant replying yes, a check must be placed on the slider to ensure it does not reach the drum too soon. Therefore if the slider is travelling too fast it must be stopped and the decision aborted (so the drum does not move). A minimum speed is also required so that the slider does not remain within the drum too long. If the speed of the slider at the start of the decision is not between the minimum and maximum speeds then the decision must be abort. If the slider controller is shared between a number of processes then resources must be allocated to ensure that insertion and retraction of the slider is possible.

Since the slider is in motion when the decision is taken, a late decision is as undesirable as an incorrect decision because even if the decision is to abort it may too late to prevent a clash. This must be prevented by including timeouts in the decision mechanism. Hence traditional commit protocols cannot be used here because they do not include such timeouts. The protocols developed in chapter 5 can be used because they have timeouts included and have also been shown to be resilient to a link failure.

To calculate timeout values, timing constraints must be placed on the decision processes. Assume initially that the slider is at its home position and the drum is in its index position, the state of the system is now safe. A decision as to the outcome of the atomic action (commit / abort) must now be made by the time the slider reaches its decision point, D_d as shown in figure (6.10). To allow enough time for message propagation and computation the decision must be started at time D_s. To achieve a timely decision the timeout values are calculated from the deadlines so that the last timeout can fire and abort the decision by D_d.

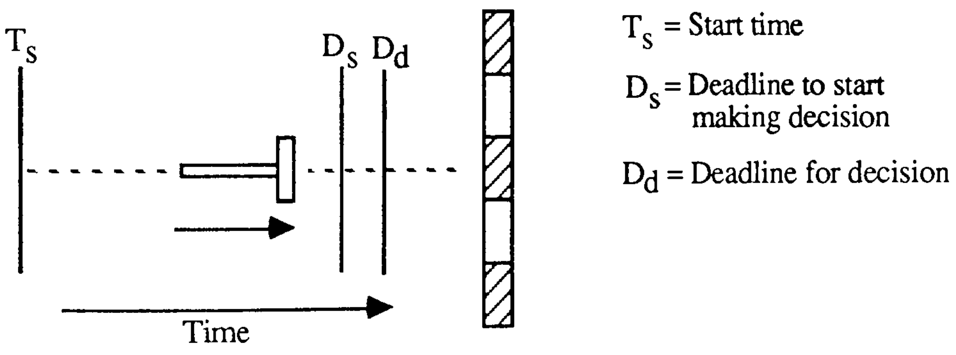

T_s
D_s D_d
Time
T_s = Start time
D_s = Deadline to start making decision
D_d = Deadline for decision

Fig.(6.10) Slider motion

As before the system can be modelled using Petri nets, these can be analysed in a fault free environment and shown to be correct. An outline Petri net of the system is shown in figure (6.11), the decisions are not shown in detail but can be assumed to produce consistent results. This means that tokens appears in P_3 and P_7 or in P_4 and P_8 but not in a combination of the two.

The system can be analysed for functionality by executing the Petri net of figure (6.11). Initially the slider is at rest (P_1) and the drum is in its safe index position (P_9). The two controllers now execute in parallel until they attempt to coordinate. The slider is accelerated until the start decision point is reached (P_2) whereupon the coordinator communicates with the participant about the decision. Meanwhile the participant has entered its decision and has been waiting for the start decision message from the coordinator. The two decisions now produce either an abort decision (P_3 and P_7) or a commit decision (P_4 and P_8). If aborted the drum is not moved and the slider is stopped. If committed the drum rotates to its new position and the slider is inserted.

Also if the actions are committed various timing constraints are guaranteed, such as the absolute firing times for t_7 and t_3 must satisfy Max (t_7) < Min (t_3). This ensures the drum will have completed rotation before the slider will be inserted, these timing constraints have been guaranteed by the commit protocol ensuring resource allocation and cannot be violated. For example the drum controller may have scheduled the processor so that it processes only the movement of the drum for the next X seconds, this is guaranteed and cannot be interrupted by any other process.

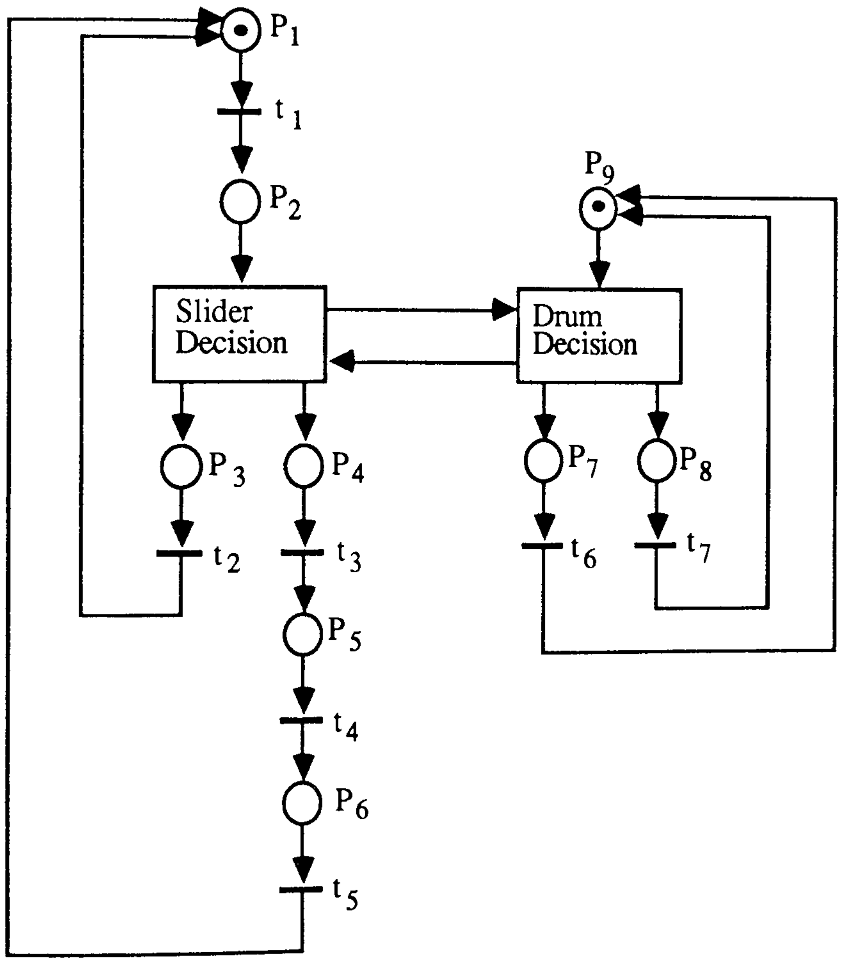


Fig.(6.11) Outline Petri net for slider and drum control

The places and transitions of figure (6.11) define the following conditions and events :-

P_1 = Slider at rest	t_1 = accelerate slider
P_2 = Slider at decision point	t_2 = Re-initialize slider
P_3 = Stop slider enabled	t_3 = Insert slider
P_4 = Slider Insert enabled	t_4 = Withdraw slider
P_5 = Slider fully inserted	t_5 = Re-initialize slider
P_6 = Retract slider enabled	t_6 = Do nothing
P_7 = Stop drum enabled	t_7 = Rotate drum
P_8 = Drum rotate enabled	
P_9 = Drum in safe position	

To examine the effect of failures on the system the specification of the decision must be included. Only link failures can be considered in this system because if a site fails then control of the system will be lost. The full Petri net model of the system is shown in figure (6.12). Note that the places P_1 -P_8 shown in figure (6.11) are the same as those in figure (6.12). However, the transitions of figure (6.11) have been renumbered for ease of analysis of the reachability tree and do not correspond directly with the transitions shown in figure (6.12). The reachability tree of figure (6.13) shows all the possible final states of the system under normal and link failure conditions. It can be seen that all the final states are consistent and also the drum rotates to its new position before the slider is inserted.

A consistent decision is always produced even if the link between the controllers fails at any time. Since messages cannot be lost (unlike a protocol using asynchronous communications) the undesirable system state P_3P_8 can never be reached, this can be verified by by examining the reachability tree. As an example consider the link failing before the participant receives the commit message, the coordinator can timeout and abort the decision safe in the knowledge that the participant will also abort. This is modelled in the Petri net of figure (6.12) by transition t_{12} failing to fire, the timer transitions tm_3 and tm_6 now fire thus providing the decision. The order in which tm3 and tm6 fire is irrelevant because the final state reached is the same, this can be verified by examining the reachability tree. A link failure is more likely to occur than site failures in real-time systems because of the mechanics being controlled. An example is a link being trapped in a robot arm joint.

This system cannot be made resilient to site failures without redundant processors because when a site fails control of the environment is lost. For example a site failure could occur after the a commit decision had been made and acted upon, for example the drum may move only partially before the controller fails. In this case there is no way a clash can be prevented because the slider will have passed D_d which is defined as the closest position to the drum where the slider can be safely stopped. It is difficult to guarantee timing constraints when site failures occur because the repair time and recovery time of a site are unpredictable. A possible approach to solve this problem is by the use of redundant processors, but this will necessarily involve extra communication for coordination.

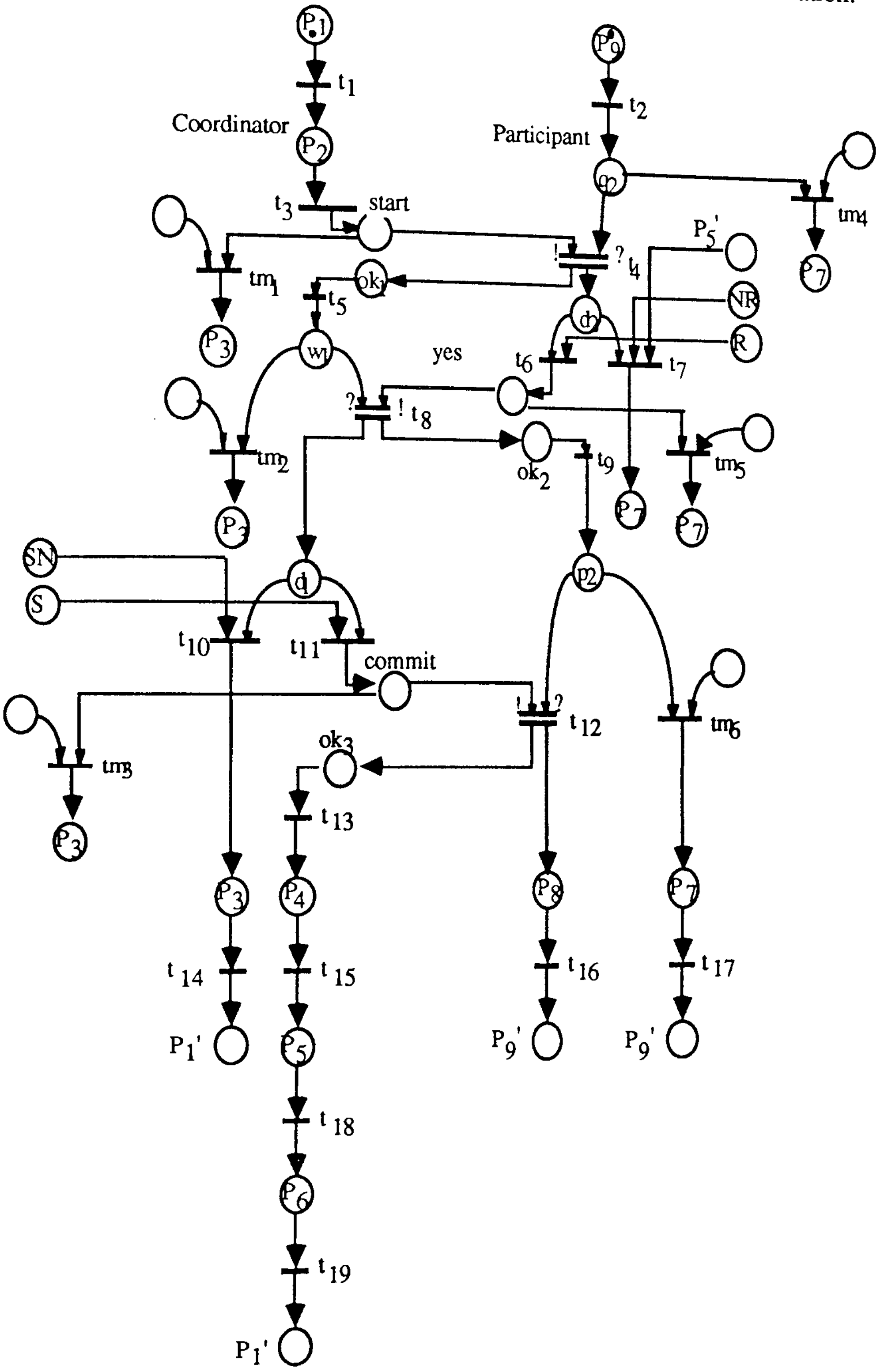


Fig.(6.12) Petri net for slider and drum control

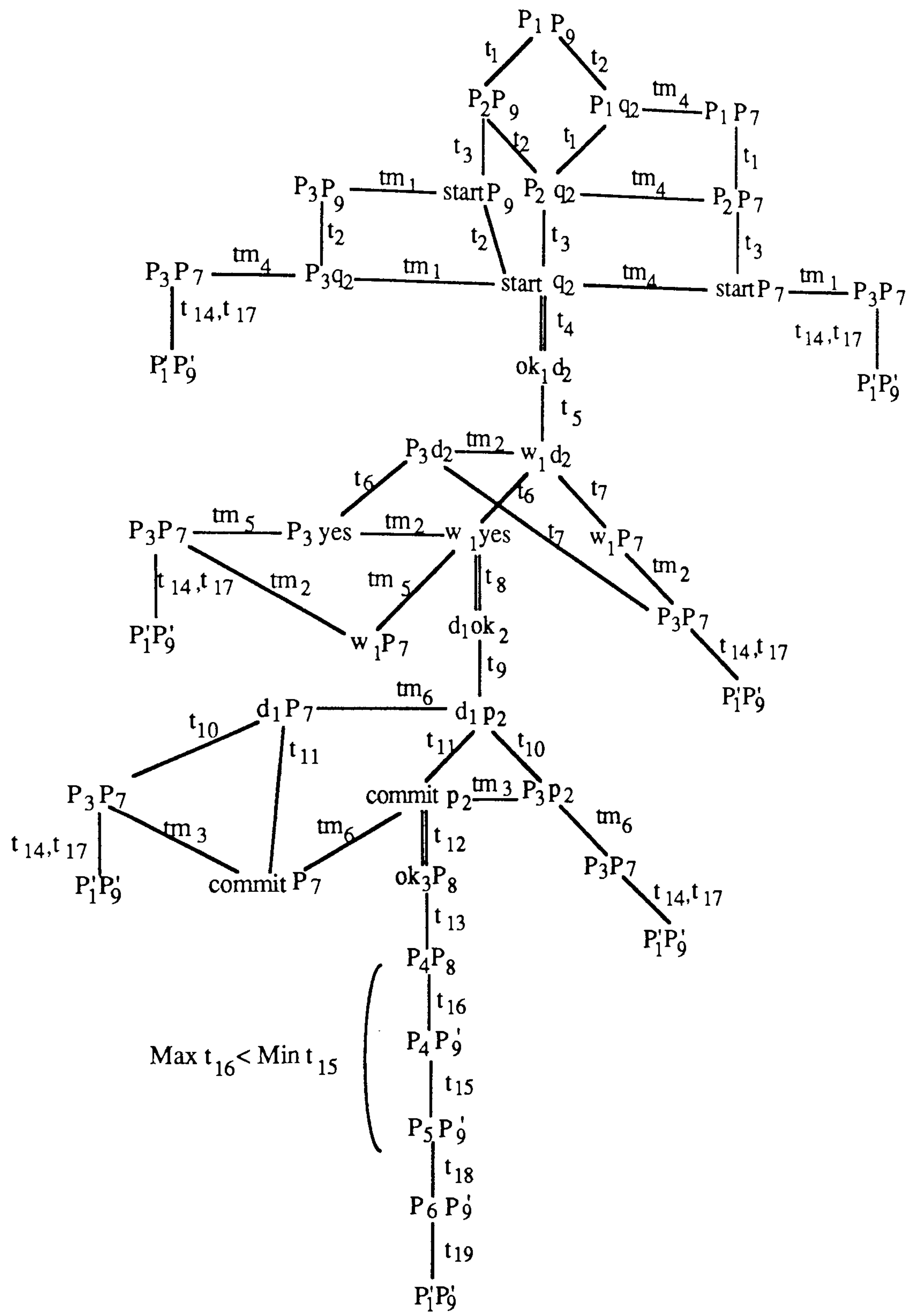


Fig.(6.13) Reduced reachability tree for fig.(6.12)

Timing constraints can be added to the Petri net in the form of minimum and maximum transition firing times. These times are the minimum and maximum times that can elapse after the transition is enabled before it fires. The times then represent upper and lower bounds on event times. These times can be converted to absolute time by adding the time the transition is enabled to the firing times.

This example requires that a decision is made at the latest by time D_d, for this to happen various timing constraints must be satisfied. If absolute times are assumed then the timing constraints applied to figure (6.12) are :-

(1) $\text{Max}(t_{12}) < D_d$

(2) $\text{Max}(tm_3) < D_d$ and $\text{Min}(tm_3) > \text{Max}(t_{12})$

(3) $\text{Max}(tm_6) < D_d$ and $\text{Min}(tm_6) > \text{Max}(t_{12})$

(4) $\text{Max}(tm_2) < D_d$ and $\text{Min}(tm_2) > \text{Max}(t_8)$

(5) $\text{Max}(tm_5) < D_d$ and $\text{Min}(tm_5) > \text{Max}(t_8)$

(6) $\text{Max}(tm_1) < D_d$ and $\text{Min}(tm_1) > \text{Max}(t_4)$

(7) $\text{Max}(tm_4) < D_d$ and $\text{Min}(tm_4) > \text{Max}(t_4)$

The first constraint ensures that if no failures occur and the decision to commit is made then the decision is computed before D_d. The constraints (2) to (7) state that the timeout transitions must fire before the deadline but after the latest time that the communications can occur. This guarantees that the communication will take place if the link has not failed.

A timer transition (eg. tm_1) is special in that an extra delay, the timer value is associated with the transition. When enabled a timer transition is assumed to start a timer, when it expires the transition may fire. The normal minimum and maximum firing times are still associated with timer transitions thus modelling timer bounds. This is realistic because precise timers cannot be implemented due to unpredictable overheads.

Timeout values have to be calculated carefully to prevent false timeouts occurring. For example consider the first communication of figure (6.12), transition t_4, this is enabled to fire when tokens are present in both the places start and q_2. The timer tm_1 is enabled as soon as a token appears in the place start, thus the value of tm_1 must be calculated so that it does not timeout before Max (t_4), even though t_4 is not enabled until a token arrives in q_2. The reverse situation is also true for the timer tm_4.

Assuming relative times for the transition times and that places P_1 and P_9 both have tokens at time 0, the earliest that tm_1 is enabled is $\text{Min}(t_1) + \text{Min}(t_3)$. Thus if the value of the timer is D_{tm_1} the transition tm_1 is able to fire after the time $\text{Min}(t_1) + \text{Min}(t_3) + D_{tm_1}$. The earliest time that tm1 can now fire is $\text{Min}(t_1) + \text{Min}(t_3) + D_{tm_1} + \text{Min}(tm_1)$. In this situation t_4 can only fire when tokens are in start and q_2, the latest that this happens is $\text{Max}[(\text{Min}(t_1) + \text{Min}(t_3)), \text{Max}(t_2)]$, let this be $L(t_4)$. The latest that t_4 fires is now $L(t_4) + \text{Max}(t_4)$ and so to allow enough time for communication the following must hold :-

$$\text{Min}(t_1) + \text{Min}(t_3) + D_{tm_1} + \text{Min}(tm_1) > L(t_4) + \text{Max}(t_4)$$

and so the timer value is found as :-

$$D_{tm_1} > L(t_4) + \text{Max}(t_4) - \text{Min}(t_1) - \text{Min}(t_3) - \text{Min}(tm_1)$$

Another constraint on D_{tm_1} is that the latest absolute time that tm_1 can fire must be less than D_d. Therefore for the relative times of t_1,t_3 and tm_1 the following must hold :-

$$\mathrm{Max}\ (t_1) + \mathrm{Max}\ (t_3) + \mathrm{Max}\ (tm_1) + D_{tm_1} < D_d$$

$$\therefore D_{tm_1} < D_d - \mathrm{Max}\ (t_1) - \mathrm{Max}\ (t_3) - \mathrm{Max}\ (tm_1)$$

The other timeout values follow in a similar manner but general expressions are difficult to calculate because of the proliferation of possible maximum enabled times.

The coordination of the slider drum system is an example of timed atomic commitment. This section has demonstrated how such coordination can be provided by using a commit protocol with bounded synchronous communications. Although it has not been implemented, the slider drum system is a good example of how timed atomic commitment can be built into a real-time system. By taking advantage of the fact that processes can operate independently when a link fails the decision produced by the protocol can be made consistent and timely. It has also been shown how upper and lower bounds for the timeout values can be calculated by examining the Timed Petri net model of the system.

One disadvantage of using commit protocols with bounded synchronous communications is that when a site fails it appears identical to the other processes as a link failure. This means the operational sites produce a decision assuming that a link has failed which may be incorrect. Other problems are due to using timeouts, firstly timeouts must be calculated so that false timeouts are rare and that timeouts do not induce further timeouts. Secondly if the start of the system is not synchronized within certain limits then it is not possible to estimate timeout values that will always satisfy their timing constraints. This suggests that the optimal protocol developed in section (5.4) is unsuitable.

6.5. Discussion

Commit protocols have traditionally been used to provide atomicity in database systems where a timely response is not important. The coordination provided by commit protocols is often needed by control systems but a timely response may be necessary. Also database systems are being used in time critical situations which require deadlines on decisions. This chapter has proposed the use of the commit protocols developed in chapter 4 in control systems and systems which require timely decisions. It has shown how a timely response can still be maintained even if failures occur.

The protocols developed have been demonstrated by way of three illustrative examples, a real-time database, a control application and a real-time control application. All three demonstrate the idea of atomic actions, the real-time applications showing how deadlines can be applied and met even in the presence of a link failure.

When a site fails it is assumed to stop processing and the repair time of the site is unknown. The real-time database example has shown how a site failure can be tolerated by replicating the data and allowing the operational sites to continue independently. This creates temporary data inconsistencies which are resolved when the failed site recovers. When considering control applications without redundant processors a site failure would mean that actuator control is lost. A site failure can only be tolerated if no timing constraints exist for the actuator. If timing constraints do exist then the failure cannot be tolerated because there is no guarantee that the site will recover in time.

A communication link failure leaves the processors with the ability to function independently but prevents any further coordination. Previous work [Skeen 83] has shown that a commit protocol cannot be designed which tolerates a link failure if messages are lost. The protocols developed in chapter 4 cannot lose messages and therefore can tolerate a link failure. The real-time database example shows how these protocols can be used to maintain consistency at a number of sites even in the presence of a link failure. Tolerating a link failure in a control application is very important because the processors can still control their actuators. The real-time control example shows how a safe system is always maintained even when a link fails.

Problems with using the protocols of chapter 4 stem from the type of communications used. The point-to-point communications prevents their use in general purpose database systems because there is no knowledge of the transactions beforehand. Real-time systems are usually more predictable and the fixed communication network should not present any problems. A problem also occurs because of the timeouts used, the timeout periods must be calculated carefully to prevent false timeouts and cascaded timeouts. A method has been proposed which shows how timeouts can be calculated from the Timed Petri net model of the system. Another problem is caused by using timeouts to detect failures, a site and link failure cannot be distinguished. That is, if a timeout occurs the site does not know if the link failed or if the other site has failed. One possible solution would be to have redundant communication links. If a site now fails to send a message it can try another link, if it still fails then it is highly probable that the receiving site has failed.

Chapter 7
Conclusions and further work

7.1. Conclusions

Computers are increasingly being used to control applications which have the potential to become dangerous if incorrectly controlled or if failures occur. Formal techniques can be used to ensure the controlling software performs as expected but failures of processing sites and communication links can still occur at any time. Such failures must be anticipated during the design stage so that mechanisms can be incorporated to ensure the system still meets its functional and/or temporal specifications.

It is often required that two or more sites need to be coordinated in such a way that either they all perform their actions or none at all. In safety critical systems it may be necessary for this functional atomicity to be provided even in the presence of failures. Also in a real-time system such atomicity may be required within specified temporal constraints.

This thesis has considered the design of protocols to achieve such atomicity and preserve consistency of actions subject to timing constraints and also in the presence of failures. The protocols have been developed for a network of Transputers where synchronous message passing is the only method of communication. Deadlines have been applied to the protocols so that they can be used to provide atomicity in real-time systems.

This research led to a number of results concerning commit protocols and their application to real-time systems. One major result is that the type of communications used to implement a protocol does affect its resiliency to failures. The extended 2 phase commit protocol has been identified as the most applicable to real-time systems, this is because when implemented using synchronous communications it can be made independently recoverable to site and communication failures.

The 2 phase commit protocol was studied and shown to remain a blocking protocol, independent of the type of communication used. The extended 2 phase commit was then investigated using asynchronous and synchronous communications. This research has proposed the placement of timeouts so that the asynchronous extended 2 phase commit can be made independently recoverable to site failures. It has also been shown that this protocol cannot tolerate a link failure if asynchronous communications are used.

However, another major contribution of this work is that the extended 2 phase commit using synchronous communications can be made independently recoverable even in the presence of a link failure. The placement of timeouts to achieve this have been suggested throughout this thesis. Similarly, the synchronous 2 phase commit can also be made independently recoverable from site failures.

These results were achieved by modelling the commit protocols using a formal model firstly in a fault-free environment and then with selected faults included. In the fault free environment it must be shown that the protocols developed satisfy certain properties such as deadlock freeness, consistency and timeliness. Previous formal models used for commit protocol specifications (FSM) were studied and found to be lacking in timing information and also did not include the communication system within the model. Without timing information the protocols cannot be analysed for timeliness and deadlines cannot be derived. The communication system was not modelled in the FSM specifications, instead a local area network was assumed to fully connect all the sites and provide facilities to deal with communication problems such as timeouts and lost messages. Since the protocols were to be implemented on a Transputer system where the communication is point-to-point the message passing must be modelled explicitly in the specifications.

This thesis uses Petri nets as a formal model for the protocol specifications because timing information can be included easily [Merlin 76, Razouk 84] and it also allows succinct modelling of point-to-point communications.

One advantage of including the communications within the model is that communication failures can be added to the Petri net so that the protocol resiliency can be investigated. From adding such failures it is found that timeouts must be included to prevent deadlock. The analysis of the Petri nets is by the use of the reachability tree which shows all possible states that can be achieved during the protocol execution. If failures are included in the Petri net specification and the Petri net is then executed the corresponding reachability tree may show a final global state which is deadlocked. From the analysis of the tree with all possible failures included it is possible to determine where timeouts should be placed and what action should be taken after timing out to preserve consistency.

Another advantage of using Petri nets as a formal model is that they can be mapped almost directly into Occam source code [Carpenter 88a]. Problems involved with mapping Petri nets into Occam are explained during the course of the protocol designs. Analysis of the protocols under failure conditions also revealed that pure Occam has insufficient semantics to prevent deadlock because output guards are not allowed in an alternate command. This problem is shown to be solved by using pre-defined Inmos assembler routines which allow both input and output to terminate after a specified time if communication is not attempted. These assembler routines have been used to implement the synchronous protocols developed in this thesis. The implementations are written in Occam and have been used for a pragmatic investigation of their resiliency to site and link failures.

The Petri net specification of the protocols were shown in chapter 4, both asynchronous and synchronous communications were included so that existing theorems for use with FSM could be investigated. This shows that the Petri net model with asynchronous communications was equivalent to the original FSM model and could be analysed using similar methods. It is shown how site failures and message loss can be modelled on the Petri net model using a uniform technique for both types of failure. It is shown that to design non-blocking protocols using synchronous communications the previous design techniques cannot be used, another method using the Petri net model is proposed. It is also shown that synchronous communications does affect the resiliency of the protocols to failures. In particular previous results show that a network partition cannot be tolerated if message loss occurs, but using synchronous communications messages cannot be lost so a single partition can be tolerated, other scenarios are also investigated.

The protocols were originally developed by replacing asynchronous communication calls directly with synchronous communication calls which produced redundant messages. After removing the redundant messages it was found that further optimizations could be obtained by assuming that lack of a message implies a negative response. For instance if a site sends a message and awaits a reply, then no reply within a specified time can be assumed to mean the other site has decided no. Of course the determination of timeouts becomes very critical because false timeouts can occur due to slow communications or slow processing. Since the communication system on a Transputer network is dedicated the communication delays can be determined accurately thus making it easier to deduce worse case timeout values.

Applications of commit protocols using synchronous communications and deadlines were proposed in chapter 6. Firstly a database example was used to demonstrate the two conflicting properties of availability and consistency of data. It is suggested that the protocols developed can be used to allow timely termination of transactions. An example of achieving data consistency when a link fails is presented. The transactions are assumed to be periodic and possess a deadline by which they must be completed. This is dealt with in the commit protocol by having a deadline by which the commit/abort decision must be made by, once this deadline has been calculated all other intermediate timing constraints can be made.

Using commit protocols in database systems to provide consistency of data is standard practice (deadlines are not). Chapter 6 also proposes their use in control applications. The need for the consistency of events as provided by a traditional commit protocol is shown by way of a level crossing example. This example shows how two tasks may need to be coordinated to retain a safe system. It is shown to be safe even in the presence of a single site failure. The next example, the control of a drum/slider mechanism shows how time can be applied to the commit protocol so that decisions can be augmented with a deadline. It is shown how timing constraints can be used to provide timely and functionally consistent decisions by analysis of a Petri net model of the system.

7.2. Further work

This thesis has been concerned with applying time constraints to commit protocols and using the resulting protocols to provide decisions in real-time systems. The principle behind a commit protocol is that of atomicity, although important this property has only been incorporated into one programming language, ARGUS [Liskov 88] and this does not allow timing constraints. It would be worth investigating how existing languages can be extended with such constructs, in particular languages designed for real-time systems should incorporate the atomicity mechanisms if they are to be used in safety critical situations.

The specification of the protocols has been carried out using Petri nets with time added to transitions. The analysis of such nets is complicated because state information is combined with timing information, automatic analysis would be advantageous and less error prone. Therefore additional work is required to develop a tool which can analyse a Petri net, the tool should be able to check for all the standard Petri net properties such as reachability and liveness. In addition, timing properties and consistency could be analysed thus proving the correctness of the Petri net. This tool could also be used during the design of a protocol by allowing automatic generation of the concurrency sets for each state.

An extension to this work would be to model and analyse the protocols using Temporal Petri nets [Suzuki 89], these were developed because certain properties such as eventuality and fairness cannot be modelled using time Petri nets. The fixed execution and delays used in time Petri nets are replaced by expressions using Temporal logic which is more precise and flexible. A previous application of these nets has been to a real-time control problem [Sagoo 90] where properties such as timeliness and safety were investigated.

As the number of sites involved in the protocol increase so does the number of states in the Petri net (known as the 'state explosion' problem) thus making analysis harder. A method which reduces the number of states in the Petri net but still keeps all the timing constraints and all other properties is worth researching.

Other work which can be considered as a continuation of this is to find further applications for the commit protocols and also to investigate the use of other database techniques in real-time systems.

One application for commit protocols could be during the reconfiguration of processor networks. During the reconfiguration of the communication links the connections must only be changed when no messages are in transit. This involves coordination between all processors before reconfiguration can occur, also if a failure of a reconfigurer occurs, after recovering the reconfigurer must bring its connections to a state consistent with the others.

Other database techniques which at first sight appear applicable to real-time systems are concurrency control and replication. Locking, timestamps and optimistic techniques have been used in database systems to provide serializability, these techniques may be more useful than strict mutual exclusion in real-time systems. As an example an optimistic concurrency control technique allows all transactions to complete, it then checks to see if any conflicts have arisen. If not the transactions are allowed and the database is updated but if conflicts do arise some transactions must be restarted, this allows greater concurrency but involves restarting a transaction from its start.

Also more work is required to design protocols which allow transactions to continue but still maintain concurrency control when partitions occur. Replication has been used in database systems to enhance availability of data but incurs extra overheads in controlling the consistency of the copies. Real-time control systems already use replicated software [Avizienis 85] to provide fault tolerance but these are used to mask design faults. Techniques similar to those used to ensure concurrency control in databases with replicated data may be of use in real-time control systems, eg. to provide backup for failed processors.

References

[Abbott 88] R.Abbott, H.Garcia-molina, "Scheduling real-time transactions", SIGMOD record, Vol.17, No.1, 1988, pp 71 - 81

[Anderson 81] T.Anderson, P.A.Lee, "Fault tolerance, principles and practice", Prentice Hall, 1981

[Anderson 85] A.Anderson, D.A.Barret, D.N.Halliwell, M.Moulding, "Software fault tolerance, an evaluation", IEEE Trans. Software Eng., Vol SE-11, No 12, 1985, pp 1502 - 1510

[Andrews 83] G.R.Andrews, F.B.Schneider, "Concepts and notations for concurrent programming", ACM Computing Surveys, Vol.15, No.1, 1983, pp 3 - 43

[Avizienis 75] A.Avizienis, "Fault tolerance and fault intolerance : complementary approaches to reliable computing", Proc. int. conf. on reliable software, ACM SIGPLAN, 1975, pp 458 - 464

[Avizienis 85] A.Avizienis, "The N-version approach to fault tolerant software", IEEE Trans. Software Eng., Vol.SE-11, No.12, 1985 pp 1491 - 1501

[Azema 76] P.Azema, R.Valette, M.Diaz, "Petri nets as a common tool for design verification and hardware simulation", Proc. 13th IEEE design automation conf., 1976, pp 106 - 116

[Bal 90] H.Bal, J.Steiner, A.S.Tanenbaum, "Programming languages for distributed computing systems", ACM Computing Surveys, Vol.21, No.3, 1988, pp 261 - 321

[Balter 81] R.Balter, "Selection of a commitment and recovery mechanism for a distributed transactional system", Proc. 1st int. symp. on Reliability in distributed software database systems, IEEE 1981, pp 21 - 26

[Basu 87] A.Basu, "Parallel procesing systems : a nomenclature based on their characteristics", IEE Proc. part I, Vol.134, No.3, 1987, pp 143 - 147

[Bernstein 80] P.A.Bernstein, D.W.Shipman, J.B.Rothnie, "Concurrency control in a system for distributed databases (SDD-1)", ACM Trans. Database Systems, Vol.5, No.1, 1980, pp 18 - 52

[Bernstein 81] P.A.Bernstein, N.Goodman, "Concurrency control in distributed database systems", ACM Computing Surveys, Vol.13, No.2, 1981, pp 186 - 221

[Bernstein 83] P.A.Bernstein, N.Goodman, V.Hadzilacos, "Recovery algorithms for database systems", in Information Processing 83, R.E.A.Mason (Editor), IFIP 1983, pp 799 - 807

[Bernstein 87a] P.A.Bernstein, V.Hadzilacos, N.Goodman, "Concurrency control and recovery in database systems", Addison-Wesley, 1987

[Bernstein 87b] P.A.Bernstein, N.Goodman, "A proof technique for concurrency control and recovery algorithms for replicated databases", Distributed Computing, Vol.2, No.1, 1987, pp 32 - 44

[Berthomieu 83] B.Berthomieu, M.Menascre, "An enumerative approach for analysing Time Petri nets", Proc. 1983 IFIP Congress, 1983, pp 41 - 46

[Bhargava 83] B.Bhargava, "Resilient concurrency control in distributed database systems", IEEE Trans. Reliability, Vol.R-32, No.5, 1983, pp 437 - 443

[Bhargava 87] B.Bhargava "Transaction processing and consistency control of replicated copies during failures in distributed databases", Journal of Management Information Sciences, Vol.4, No.2, 1987, pp 93 - 112

[Bloch 89] G.Bloch, I.M.Macleod, "An analysis of error recovery problems in distributed computer control systems", Proc. 8th IFAC workshop on distributed computer control systems, 1988, pp 81 - 85

[Bornat 86] R.Bornat, "A protocol for generalized Occam", Software Practise and Experience, Vol.16, No.9, Sep.1986, pp783 - 799

[Brilliant 89] S.S.Brilliant, J.C.Knight, N.G.Leveson, "The consistent comparison problem in N-version software", IEEE Trans. Software Eng., Vol.SE-15, No.11, 1989, pp 1481 - 1485

[Burns 90] A.Burns, A.Welling, "Real-time systems and their programming languages", Addison-Wesley, 1990

[Campbell 86] R.H.Campbell, B.R.Randell, "Error recovery in asynchronous systems", IEEE Trans. Software Eng., Vol.SE-12, No. 8, 1986, pp 811 - 826

[Carpenter 87] G.F.Carpenter, "The use of Occam and Petri nets in the simulation of logic structures for the control of loosely coupled distributed systems", in Proc. UKSC conf. on computer simulation, 1987, pp 30 - 35

[Carpenter 88a] G.F.Carpenter, D.J.Holding, A.M.Tyrrell, "The analysis and protection of interprocess communication in real-time systems", Journal of IERE, Vol. 58, No. 4, June 1988, pp 173 - 180

[Carpenter 88b] G.F.Carpenter, D.J.Holding, A.M. Tyrrell, "The design and simulation of software fault tolerant mechanisms for application in distributed processing systems", Microprocessing and Microprogramming Vol.22, 1988, pp 175 - 185

[Ceri 87] S.Ceri, G.Pellagati, "Distributed databases, principles and systems", McGraw Hill, 1987

[Chan 83] A.Chan, U.Dayel, S.Fox, N.Goodman, D.R.Ries, D.Skeen, "Overview of an ADA compatible distributed database manager", ACM SIGMOD Record, Vol.13, No.4, May 1983, pp 228 - 237

[Chin 87] F.Y.Chin, K.V.S.Ramarao, "Information based model for failure handling", IEEE Trans. Software Eng., Vol SE-13, No.4, 1987, pp 420 - 437

[Chull 89] M.E.Chull, A.Zarea-alibadi, "Real-time system implementation - the Transputer and Occam alternative", Microprocessing and Microprogramming, Vol.26, 1989, pp 77 - 84

[Coolahan 83] J.E.Coolahan, N.Roussoploulos, "Timing requirements for time driven systems using augmented Petri nets", IEEE Trans. Software Eng., Vol.SE-9, No.5, 1983, pp 603 - 616

[Davidson 85] S.B.Davidson, H.Garcia-molina, D.Skeen, "Consistency in partitioned networks", ACM Computing Surveys, Vol.17, No.3, Sep. 1985, pp 341 - 370

[Davidson 89] S.Davidson, I.Lee, V.Wolfe, "Language constructs for timed atomic commitment", Proc. 19th int. symp. on fault tolerant computing, IEEE 1989, pp 470 - 477

[Dijkstra 68] E.W.Dijkstra, "Co-operating sequential processes", From Programming languages F.Genuys (Ed.), Academic Press, 1968

[Dijkstra 72] E.W.Dijkstra, "Notes on structured programming", Academic Press, 1972

[Dijkstra 75] E.W.Dijkstra, "Guarded commands, non-determinacy and a calculus for derivation of propositions", Proc. int. conf. on reliable software, ACM SIGPLAN, 1975, pp 2.0 - 2.13

[Dod 83] Department of defense (USA), "Reference manual for the Ada programming language", ANSI/MIL-STD-1815A, Dod, Washington D.C.,1983

[Dwork 83] C.Dwork, D.Skeen, "The inherent cost of non-blocking commitment", Proc. symp. principles of distributed computing, ACM 1983, pp 1 - 11

[Eswaren 76] K.P.Eswaren, J.N.Gray, R.A.Lorie, I.L.Traiger, " The notions of consistency and predicate locks in a database system", ACM Communications, Vol.19, No.11, Nov.1976, pp 624 - 633

[Fleming 87] P.J.Fleming (editor), "Parallel processing in control - the Transputer and other architectures", Peter Peregrinus Ltd, 1988

[Flynn 66b] M.J.Flynn, "Very high speed computing systems", Proc. IEEE, Vol.54, 1966, pp 1901 - 1909

[Gheith 89] A.Gheith, K.Schwan, "CHAOS^{ART} : Support for real-time transactions", Proc. 19th int. symp. on Fault-tolerant computing, IEEE, 1989, pp 462 - 469

[Gray 79] J.N.Gray, "Notes on database operating systems", in Operating systems : an advanced course, Springer-Verlag 1979, pp 393 - 481

[Gregory 85] S.T.Gregory, J.C.Knight, "A new linguistic approach to backward error recovery", Digest of papers FTCS-15, 15th annual symposium on Fault tolerant computing, IEEE, 1985, pp 404 - 409

[Gregory 89] S.T.Gregory, J.C.Knight, "On the provision of backward error recovery in production programming languages", Proc. 19th international symposium on Fault tolerant computing, IEEE, 1989, pp 506 - 511

[Haerder 83] T.Haerder, A.Reuter, "Principles of transaction oriented database recovery", ACM Computing Surveys, Vol 15, No 4, Dec 1983, pp 287 - 317

[Halici 89] U.Halici, A.Dogac, "Concurrency control in distributed databases through time intervals and short term locks", IEEE Trans. Software Eng., Vol.15, No.8, 1989, pp 994 - 1005

[Hammer 80] M.Hammer, D.Shipman, "Reliability mechanisms for SDD-1 : A system for distributed databases", ACM Trans. Database Systems, Vol.5, No.4, 1980, pp 431 - 466

[Haskin 88] R.Haskin, Y.Malachi, N.Sawdon, G.Chan, "Recovery management in Quicksilver', ACM Trans. Compting Systems, Vol.6, No.1, Feb 1988, pp 82 - 108

[Hecht 76] H.Hecht, "Fault tolerant software for real-time applications", ACM Computing Surveys, Vol.8, No.4, 1976, pp 391 - 407

[Hecht 79] H.Hecht, "Fault tolerant software", IEEE Trans. Reliability, Vol.R-28, No.3, 1979, pp 227 - 232

[Hoare 74] C.A.R.Hoare, "Monitors: an operating system structuring concept", ACM Communications, Vol.17, No.10, 1974, pp 549 - 557

[Hoare 78] C.A.R.Hoare, "Communicating sequential processes", ACM Communications, Vol.21, No.8, 1978, pp 666 - 677

[Holding 88] DJ.Holding, M.R.Hill, G.F.Carpenter, "The design of distributed, software fault tolerant, real-time systems incorporating decision mechanisms", Microprocessing and Microprogramming, Vol.24, 1988, pp 801 - 806

[Hwang 84] K.Hwang, F.A.Briggs, "Computer architectures and parallel processing", McGraw-Hill, 1984

[Inmos 88a] Inmos Ltd, "Occam 2 reference manual", Prentice Hall, 1988

[Inmos 88b] Inmos Ltd., "Transputer reference manual", Prentice-Hall, 1988

[Inmos 89] Inmos Ltd., "Transputer development system", Prentice Hall, 1989

[Irwin 90] G.W.Irwin, P.J.Fleming (Eds), "Parallel processing for real-time control", Special edition IEE Proc. D, Vol 137 No.4, 1990

[Joseph 89] M.Joseph, A.Goswami, "Formal description of real time systems: a review", Information and Software Technology, Vol.31, No.2, 1989, pp 67 - 76

[Karp 69] R.Karp, R.Miller, "Parallel program schemata", Journal of Computer Systems and Science, Vol.3, No.4, 1969, pp 167 - 195

[Keller 76] R.M.Keller, "Formal verification of parallel programs", ACM Communications, Vol.7, 1976, pp 371 - 384

[Kim 88] K.H.Kim, "Programmer transparent coordination of recovering concurrent processes : Philosophy and rules for implementation", IEEE Trans. Software Eng., Vol.14, No.6, 1988, pp 810 - 821

[Knight 86] J.C.Knight, N.G.Leveson, "An experimental evaluation of the assumption of independence in multi-version programming", IEEE Trans. Software Eng., Vol.SE-12, No.1, 1986, pp 96 - 109

[Kopetz 83] H.Kopetz, "Software reliability", Macmillan Press Ltd., 1979

[Kung 81] H.T.Kung, J.T.Robinson, "On optimistic methods for concurrency control", ACM Trans. Database Systems, Vol.6, No.2, 1981, pp 213 - 226

[Lala 85] P.K.Lala, "Fault tolerant and fault testable hardware design", Prentice-Hall, 1985

[Lamport 78] L.Lamport, "Time, clocks and the ordering of events in a distributed system", ACM Communications, Vol.21, No.7, 1978, pp 558 - 565

[Lamport 82] L.Lamport, R.Shostak, M.Pease, "The Byzantine Generals problem", ACM Trans. Programming Languages and Systems, Vol.4, No.3, 1982, pp 382 - 401

[Lamport 83] L.Lamport, "Specifying concurrent program modules", ACM Trans. Programming Languages and Systems, Vol.5, No.2, 1983, pp 190 - 223

[Lee 85] I.Lee, V.Gehlot, "Language constructs for distributed real-time programming", Proc. real-time symposium, IEEE, 1985, pp 57 - 66

[Lee 89a] I.Lee, R.B.King, R.P.Paul, "A predictable real-time kernel for distributed multi-sensor systems", IEEE Computer, Vol.22, No.6, 1989, pp 78 -83

[Lee 89b] I.Lee, S.B.Davidson, V.Wolfe, "Maintaining consistency over a network in real-time applications", Proc. 1989 American control conf., IEEE, 1989, pp 528 - 533

[Leppalla 87] K.Leppalla, "Utilization of parallelism in Transputer based real-time control systems", Microprocessing and Microprogramming, Vol.21, 1987, pp 629 - 636

[Leveson 83] N.G.Leveson, "Software fault tolerance: the case for forward error recovery", Proc. 4th American Inst. astronautics and aeronautics (AIAA) conf. on computers in aerospace, 1983, pp 50 - 54

[Leveson 84] N.G.Leveson, "Software safety in computer controlled systems", IEEE Computer, 1984, pp 48 - 55

[Leveson 86] N.G.Leveson, "Software safety : Why, what and how ?", ACM Computing Surveys, Vol 18, No 2, June 1986, pp 125 - 163

[Leveson 87] N.G.Leveson, J.L.Stolzy, "Safety analysis using Petri nets", IEEE Trans. Software Eng., Vol.SE-13, No.3, 1987, pp 386 - 397

[Lin 88] K.J.Lin, M.J.Lin, "Enhancing availability in distributed real-time databases", ACM SIGMOD Record, Vol.17, No.1, 1988, pp 34 - 43

[Lindsay 82] B.G.Lindsay, L.M.Haas, C.Mohan, P.F.Wilms, R.A.Yost, "Computation and communication in R* : a distributed database manager", Operating Systems Review, Vol.17, No.5, 1983, pp 1- 2

[Liskov 88] B.Liskov, R.Scheifler, "Guardians and actions: linguistic support for robust distributed programs", Proc. 9th symp on Principles of Programming Languages, 1982, pp 7 - 19

[London 75] R.L.London, "A view of program verification", Proc. int. conf. on reliable software, 1975, pp 534 - 545

[Mancini 89] L.V.Mancini, S.K.Shrivastava, "Replication within atomic actions and conversations : a case study in fault tolerance duality", Proc. 19th int. symp. Fault tolerant Computing, IEEE, 1989, pp 454 - 461

[May 85] D.May R.Shepherd, "Occam and the Transputer", in Concurrent languages in distributed systems, G.L.Reijns and E.I.Dagless (editors), IFIP 1985, pp 19 - 33

[Mekly 80] L.J.Mekly, S.S.Yau, "Software design representation using abstract process networks", IEEE Trans. Software Eng., Vol.SE-6, No.9, 1980, pp 420 - 434

[Memmi 84] G.Memmi, P.Behm, "RAFAEL : a real-time system analysis tool", Proc. 2nd conf. on software eng. practice and experience, 1984, pp 1 - 7

[Merlin 76] P.M.Merlin, D.J.Farber, "Recoverability of communication protocols - implications of a theoretical study", IEEE Trans. Communications, Vol.24, No.7, 1976, pp 1036 - 1043

[Merlin 78] P.M.Merlin, B.Randell, "State restoration in distributed systems", Digest of papers FTSC-8, 8th annual international symposium on Fault tolerant computing, IEEE, 1978, pp 129 - 134

[Mohan 83a] C.Mohan, B.Lindsay, "Efficient commit protocols for the tree of processes model of distributed transactions", Proc. 2nd SIGACT/SIGOPS symp. on distributed computing, ACM 1983, pp 76 - 88

[Mohan 83b] C.Mohan, R.Strong, S.Finkelstein, "Method for distributed transaction commit and recovery using Byzantine agreement within a cluster of processes", Proc. 2nd SIGACT/SIGOPS symp. on distributed computing, ACM 1983, pp 89 - 103

[Mohan 85] C.Mohan, "Lock conversion in Non-two-phase locking protocols", IEEE Trans. Software Eng., Vol.SE-11, No.1, 1985, pp 15 - 22

[Moitra 90] A.Moitra, M.Joseph, "Implementing real-time systems by transformations", in Real-time systems, theory and applications, H.M.S.Zedan (Editor), Elsevier science publishers B.V. (North-Holland), 1990, pp 143 - 157

[Moser 90] L.E.Moser, P.M.Melliar-Smith, "Formal verification of safety critical systems", Software Practice and Experience, Vol.20, No.8, 1990, pp 799 - 821

[Murata 89] T.Murata, "Petri nets : Properties, analysis and applications", Proc. IEEE, Vol.77, No.4, 1989, pp 541 - 580

[Nelson 83] R.A.Nelson, L.M.Haibt, P.B.Sheridan, "Casting Petri nets into programs", IEEE Trans. Software Eng., Vol.SE-9, No.5, 1983, pp 590 - 602

[Neumann 85] P.G.Neumann, "Some computer related disasters and other egregious horrors", ACM SIGSOFT, Software Engineering Notes, Vol 10, no 1, Jan 1985, pp 6 -11

[Parnas 77] D.L.Parnas, "The use of precise specifications in the development of software", Proc. IFIP congress, 1977, pp 861 - 867

[Peterson 81] J.L.Peterson, "Petri net theory and the modelling of systems", Prentice Hall, 1981

[Ramamritham 84] K.Ramamritham, J.A.Stankovic, "Dynamic task scheduling in hard real-time systems", IEEE Software, Vol.1, No.3, 1984, pp 65 - 75

[Ramchandani 74] C.Ramchandani, "Analysis of asynchronous concurrent systems by Petri nets", Phd thesis, project MAC, report no. MAC-TR-120, MIT, Cambridge, 1974

[Randell 75] B.R.Randell, "System structure for software fault tolerance", IEEE Trans. Software Eng., Vol SE-1, No 2, June 1975, pp220 - 232

[Randell 78] B.Randell, P.A.Lee, P.A.Treleaven, "Reliability issues in computing system design", ACM Computing Surveys, Vol.10, No.2, 1978, pp 123 - 165

[Razouk 84] R.R.Razouk, The derivation of performance expressions for communication protocols from timed Petri nets", Computer Communication Review (USA), Vol.14, No.2, 1984, pp 210 - 217

[Razouk 85] R.R.Razouk, C.P.Phelps, "Performance analysis using timed Petri nets ", in Protocol specification, testing and verification iv, Y.Yemmi, R.Strom, S.Yemini (Eds), New York, Elsevar, 1985, pp 561 - 576

[Razouk 85] R.R.Razouk, D.S.Hirschberg, "Tools for efficient analysis of concurrent software systems", Proc. 2nd conf. on software development tools, techniques and alternatives, IEEE, 1985, pp 192 -198

[Roscoe 88] A.W.Roscoe, C.A.R.Hoare, "The laws of Occam programming", Theoretical Computer Science, Vol.60, No.2, 1988, pp 177 - 229

[Sagoo 90] J.S.Sagoo, D.J.Holding, "The specification and design of hard real-time systems using time and temporal Petri-nets", Microprocessing and Microprogramming, Vol.30, No.1 - 5, 1990, pp 389 - 396

[Schneider 87] F.B.Schneider, "The fail-stop processor approach", in Concurrency control and reliability in distributed systems, B.Bhargava (Editor), Van Nostrand Reinhold, 1987, pp 370 - 394

[Shepherd 87] R.Shepherd, "Extraordinary use of Transputer links", Inmos Technical note no. 1, March 1987

[Sifikis 77] J.Sifikis, "Use of Petri nets for performance evaluation", Proc. 3rd int. symp. on modelling and performance evaluation of computer systems, 1977, pp 75 - 93

[Singhal 88] M.Singhal "Issues and approaches to design of real-time database systems", SIGMOD Record, Vol.17, No.1, March 1988, pp 19 - 33

[Skeen 81a] D.Skeen, "Non blocking commit protocols", Proc. SIGMOD int. conf. on management of data, 1981, pp 133 - 142

[Skeen 81b] D.Skeen, "A decentralized termination protocol", Proc. of symp. on reliability in distributed software and database systems, 1981, pp 27 - 32

[Skeen 82a] D.Skeen, "Crash recovery in a distributed database system", Phd thesis, Dept. elect. eng. comput. sci., Univ. Calif., Berkeley, 1982

[Skeen 82b] D.Skeen, "A quorum based commit protocol", Proc. 6th Berkeley workshop on distributed management and computer networks, 1982, pp 69 - 81

[Skeen 83] D.Skeen, M.Stonebraker, "A formal model of crash recovery in a distributed system", IEEE Trans. Software Eng., Vol.SE-9, No.3, May 1983, pp 219 - 228

[Sommerville 85] I.Sommerville, "Software Engineering", Addison-Wesley, 1985

[Son 87] S.H.Son, "Using replication for high performance database support in distributed real-time systems", Proc of the real-time system symposium, IEEE, 1987, pp 79 - 86

[Son 88] S.H.Son, "Semantic information and consistency in distributed real-time systems", Information and Software Technology, Vol.29, No.8, 1987, pp 440 - 449

[Spivey 89] J.M.Spivey, "The Z notation : a reference manual", Prentice Hall, 1989

[Stankovic 88a] J.A.Stankovic, "Misconceptions about real-time computing : a serious problem for next generation computing systems", IEEE Computer, Oct. 1988, pp 10 - 29

[Stankovic 88b] J.A.Stankovic, W.Zhao, "On real-time transactions", ACM SIGMOD Record, Vol.17, No.1, 1988, pp 4 - 18

[Stonebraker 87] M.Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES", IEEE Trans. Software Eng., Vol.SE-5, No.3, pp 188 - 194

[Suzuki 89] I.Suzuki, L.Harngdar, "Temporal Petri nets and their applications to modelling and analysis of a handshake daisy driven arbiter", IEEE Trans. Computers, Vol.38, No.5, May 1989, pp 696 - 704

[Suzuki 90] I.Suzuki, S.M.Shatz, T.Murata, "A protocol modelling and verification approach based on a specification language and Petri nets", IEEE Trans. Software Eng., Vol.16, No.5, May 1990, pp 523 - 537

[Taylor 86] D.J.Taylor,"Concurrency and forward recovery in atomic actions", IEEE Trans. Software Eng., Vol.SE-12, No.1, 1986, pp -69 -78

[Tyrrell 87] A.M.Tyrrell, D.J.Holding, "Design of reliable software in distributed systems using the conversation scheme", IEEE Trans. Software Eng., Vol.SE-12, No.9, 1986, pp 921 - 928

[Wu 89] J.Wu, E.B.Fernandez, "A simplification of a conversation design scheme using Petri nets", IEEE Trans. Software Eng., Vol.15, No.5, 1989, pp 658 - 660

[Yuan 89] S.M.Yuan, P.Jalote, "Fault tolerant commit protocols", Proc. 5th int. conf. on data engineering, IEEE comput. soc., 6 - 10th Feb., 1989, pp 280 - 286

[Zuberek 80] W.M.Zuberek, "Timed Petri nets and preliminary performance evaluation", 7th annual symp. on computer architecture, 1980, pp 85 - 96

Appendix A

Published papers

(1) D.J.Holding, M.R.Hill, G.F.Carpenter, "The design of distributed, software fault tolerant, real-time systems incorporating decision mechanisms", Microprocessors and Microprogramming, Vol.24, 1988, pp 801 - 806

(2) M.R.Hill, D.J.Holding, "The modelling, simulation and analysis of commit protocols in distributed computing systems", Proc. UKSC conf. , Brighton, Sep. 1990

North-Holland
Microprocessing and Microprogramming 24 (1988) 801–806

THE DESIGN OF DISTRIBUTED, SOFTWARE FAULT TOLERANT, REAL-TIME SYSTEMS INCORPORATING DECISION MECHANISMS

D. J. Holding, M. R. Hill and G. F. Carpenter

Aston University, Aston Triangle,
Birmingham B4 7ET, United Kingdom.

This paper considers the use of software fault tolerance in the design of loosely-coupled real-time distributed systems. It also addresses the problem of taking distributed multi-party decisions in decentralised systems and shows that distributed database techniques can be used to provide a general solution. In particular, it is shown that a two phase locking mechanism can be used to serialise concurrent decisions, thus removing certain timing problems, and a two phase commit protocol can be used to implement distibuted decisions which are recoverable for certain classes of failure. The techniques described are illustrated by reference to a safety critical application. A solution is proposed comprising an Occam program which can be implemented on microprocessor and transputer based systems.

that they do not lead to failure [2], [3]. It can also be used to
prote
impo

This
loose
inter-
fault
tolera
syste
distril
show
to pro
to tin
a dis

complex because an error can migrate between processes
through inter-process actions which can not be revoked. The
in such systems is
ay result in the
le domino effect).
ehaviour of
ation studies are
ss actions are
using
l, interprocess
refully controlled
olerance can be
ncurrent recovery
s have been

THE MODELLING, SIMULATION, AND ANALYSIS OF COMMIT PROTOCOLS IN DISTRIBUTED COMPUTING SYSTEMS

Martin R. Hill and David J. Holding
Information Technology Research Group
Aston University
Birmingham B4 7ET, U.K.

ABSTRACT

Commit protocols are used to provide atomicity and ensure consistency in distributed computer systems. This paper considers the design and simulation of robust protocols for distributed systems which communicate by synchronous and asynchronous message passing. The protocols are modelled, simulated and analysed under normal and fault conditions using Petri nets. The state reachability trees generated from these models can be used in the design of robust protocols. The paper describes the design of a robust commit protocol for systems which use synchronous communications and are subject to site failure. This protocol has been translated into the concurrent programming language Occam and implement on a loosely coupled distributed processing system comprising a network of Transputers.

INTRODUCTION

To ensure consistency over a distributed computer system, commit protocols must be used to provide atomicity when computing distributed decisions or transactions. Such protocols need to be resilient to processor or communication failure. Traditionally

communications. The Petri nets models can be used in the verification of the protocols. Also, using known techniques, they can be extended to accommodate models of timing constraints and failure mechanisms. The paper describes how such models can be used in the design of robust synchronous commit protocols for systems subject to site failure. The paper also described how these models can be translated into the concurrent programming language Occam to produce executable models or simulations.

The technique is demonstrated by considering the design of an extended two phase commit protocol using synchronous communication. A robust version of the protocol is developed and it is shown that additional

multi-party commit protocols is similar to solving the

waiting for the failed site to recover) [Skeen 81]. The two phase commit protocol described above is a blocking protocol. While it is correct under normal conditions and does not block when a participant site fails (because upon recovery the failed site will terminate consistently but with extra communications), it may block when the coordinator site fails. Specifically, if the coordinator fails before sending a decision to the participant then the protocol must hold all its resources until the failure is repaired.

Appendix B

Occam source code for the optimized extended 2 phase commit protocol

This appendix contains the Occam source code for the following protocols using synchronous communications :

(1) 2 phase commit (see figure 4.7)

(2) Extended 2 phase commit (see figure 4.17)

(3) Optimized extended 2 phase commit (see figure 5.3).

The timeouts shown are placed for expected link failure but the protocols can easily be modified to investigate site failures. The timeouts would be placed as suggested in chapter 4. A site failure is simulated by executing a recovery procedure at the failure point and commenting out all the code after that. This recovery procedure then attempts to bring the recovering site to a state consistent with the other sites using local information.

```
-----------------------------------------------------------------------
-- 2 phase commit protocol subject to link failures
--
--
-- see 7/11/89 for details
-- Coordinator runs in slot 1
-- Participant 0 runs on slot 0
-- Particiapnt 1 runs on slot 2
-- Multiplexor runs on slot 3
--
--
-- dodgy link is between coordinator link 3
-- and participant 1 link 3
--
-- i.e. flying lead between EDGE 0 and EDGE 4
-----------------------------------------------------------------------

------------------------------------------------------------------------------
-- The network is configured as a tree of 1 coordinator 2 participants and
-- a multiplexor process
-- The multiplexor collects debug information from the coordinator and
-- participants and sends it to the host via link 2
--
-- The multiplexor runs on slot 3
-- The coordinator runs on slot 1
-- Participant 0 runs on slot 0
-- Participant 1 runs on slot 2
------------------------------------------------------------------------------

------------------------------------------------------------
-- Channels to host transputer
------------------------------------------------------------
CHAN OF ANY Debug.from.host, Debug.to.host :

------------------------------------------------------------
-- Debug channels
------------------------------------------------------------
CHAN OF ANY Debug.from.p0, Debug.to.p0 :
CHAN OF ANY Debug.from.p1, Debug.to.p1 :
CHAN OF ANY Debug.from.coord, Debug.to.coord :

------------------------------------------------------------
-- Network channels between coordinator and participants
------------------------------------------------------------
CHAN OF ANY Coord.to.p0, Coord.from.p0 :
CHAN OF ANY Coord.to.p1, Coord.from.p1 :

VAL link0in IS 4 :
VAL link0out IS 0 :
VAL link1in IS 5 :
VAL link1out IS 1 :
VAL link2in IS 6 :
VAL link2out IS 2 :
VAL link3in IS 7 :
VAL link3out IS 3 :

PROC coordinator (CHAN OF ANY From.p0, To.p0, Link.from.p1, Link.to.p1,
                                  To.debug, From.debug)
  ------------------------------------------------------------------------
  -- This procedure communicates with participants 0 and 1 and also sends debug
     info to the host transputer via the multiplexor
  --
  -- The link between the coordinator and participant 1 is the one that can
     be removed to demonstrate the resiliency
                    ---------------------------------------------------
  #USE "\dsl\lib\userio.tsr"
  #USE "\dsl\lib\interf.tsr"
  #USE "2pc.lib.tsr"
```

```
CHAN OF ANY pl.disconnected :
----------------------------------------------------------------------------
-- normal channels for reply messages from particpants
----------------------------------------------------------------------------
[no.of.participants]CHAN OF ANY part.answered :
----------------------------------------------------------------------------
-- channels to multiplex strings to the screen
----------------------------------------------------------------------------
[3] CHAN OF ANY send.to.screen :
CHAN OF ANY stop.screen.mux :
----------------------------------------------------------------------------
-- channels for decision
----------------------------------------------------------------------------
CHAN OF ANY vote :

#USE "\tdsiolib\userhdr.tsr"
#USE "\tdsiolib\slice.tsr"
#USE "\tdsiolib\filerhdr.tsr"
PROC multiplexor ([]CHAN OF ANY screen.in, CHAN OF ANY screen.out,
                         CHAN OF INT stopper)
-- multiplexes a collection of scrstream output channels onto a
-- single such channel.
-- Each change of input directs output to a new line on the screen.
-- Any input on stopper terminates the multiplexer.
  VAL n IS SIZE screen.in:
  BOOL going :
  INT tt.char :
  BYTE tt.bcom :
  INT any :
  INT current.source :
  PROC check.current.source(INT prev, VAL INT new)
  -- sends *c*n to screen.out if new<>prev, resets prev to be new
    SEQ
      IF
        new = prev
          SKIP
        TRUE
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.out.byte; BYTE ((INT'0') + new)
            screen.out ! tt.out.byte; '>'
            prev := new
  :
  SEQ
    going := TRUE
    current.source := -1
    WHILE going
      PRI ALT
        stopper ? any
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'; tt.endstream
            going := FALSE
        ALT i = 0 FOR n
          screen.in[i] ? tt.bcom
            IF
              tt.bcom = tt.out.string
                [max.string.size]BYTE str :
                INT len:
                SEQ
                  input.len.bslice (screen.in[i], len, str)
                  --check.current.source(current.source, i)
                  screen.out ! tt.out.string
                  output.len.bslice (screen.out, len, str)
              (tt.bcom = tt.out.byte) OR
               (tt.bcom = tt.ins.char)
                BYTE ch :
                SEQ
                  screen.in[i] ? ch
                  --check.current.source(current.source, i)
                  screen.out ! tt.bcom; ch
              tt.bcom = tt.out.int
                INT x :
                SEQ
                  screen.in[i] ? x
                  --check.current.source(current.source, i)
                  screen.out ! tt.out.int; x
              tt.bcom = tt.goto
                INT x,y:
                SEQ
                  screen.in[i] ? x;y
                  --check.current.source(current.source, i)
                  screen.out ! tt.goto; x; y
              tt.bcom = tt.endstream
                SEQ
                  --check.current.source(current.source, i)
                  screen.out ! tt.endstream
              TRUE
                screen.out ! tt.bcom
:
#USE "2pclib.tsr"            -- commen messages etc
PROC collect.answers (CHAN OF ANY From.p0, From.p1,
                      [no.of.participants] CHAN OF ANY P.replied,
                      CHAN OF ANY Return.vote, Duff.link.p1, To.screen)


  INT vote :
  [no.of.participants] INT Answer.from :
  INT error.detected :

  SEQ
    PAR i = 0 FOR no.of.participants
      Answer.from [i] := not.replied
    PAR
      SEQ
        From.p0 ? Answer.from [p0]
        P.replied [p0] ! Answer.from [p0]
      ALT
        From.p1 ? Answer.from [p1]
          P.replied [p1] ! Answer.from [p1]
        Duff.link.p1 ? error.detected
          SEQ
            -- link failure occurred sending message
            -- assume answer to be aam
            Answer.from [p1] := aam
            P.replied [p1] ! Answer.from [p1]
    vote := commit
    PAR i = 0 FOR no.of.participants
      IF
        Answer.from [i] = aam
          vote := abort
        TRUE
          SKIP
    Return.vote ! vote
:
#USE "2pclib.tsr"            -- commen messages etc
PROC decision ([no.of.participants]CHAN OF ANY partic.answered, CHAN OF ANY
                decision.made, Out.to.p0, Out.to.p1, stop.mux, Duff.link.p1,
                To.screen)

  #USE "\tdsiolib\userio.tsr"
  #USE "\tds2lib\reinit.tsr"    -- extraordinary library

  INT mess :
  []BYTE mess.array RETYPES mess :
  INT coord.state :
  BOOL coord.willing :
  [2]INT has.replied :
  INT vote :
```

```
TIMER time :
INT time.now :
VAL ticks.per.sec IS 15625 :
VAL time.out IS (10 * ticks.per.sec) :
INT next.tick :
VAL allowed.out.time IS (10 * ticks.per.sec) :
BOOL Link.error :
PROC send.state (CHAN OF ANY sink, VAL INT state)
  IF
    state = initial
      write.full.string (sink, " Initial ")
    state = ready
      write.full.string (sink, " Ready ")
    state = undecided
      write.full.string (sink, " Undecided ")
    state = aborted
      write.full.string (sink, " Aborted ")
    state = committed
      write.full.string (sink, " committed ")
:
PROC get.next.tick (INT next.tick, VAL INT delta)
  TIMER time :
  SEQ
    time ? next.tick
    next.tick := next.tick PLUS delta
:


SEQ
  coord.willing := TRUE
  -- initial state
  -- send prepare message to all participants
  coord.state := initial
  write.full.string (To.screen, "coord state -")
  send.state (To.screen, coord.state)
  PAR
    Out.to.p0 ! pm
    SEQ
      mess := pm
      get.next.tick (next.tick, allowed.out.time)
      OutputOrFail.t (Out.to.p1, mess.array, time, next.tick, Link.error)
      IF
        Link.error
          SEQ
            write.full.string (To.screen, "error sending pm to p1")
            Duff.link.p1 ! error
            SKIP
        NOT Link.error
          SEQ
            write.full.string (To.screen, "pm sent ok to p1")
            SKIP
  coord.state := undecided
  write.full.string (To.screen, "coord state -")
  send.state (To.screen, coord.state)
  PAR i = 0 FOR no.of.participants
    has.replied [i] := not.replied

  time ? time.now  -- start timer for answers
  PAR
    PAR i = 0 FOR no.of.participants
      partic.answered [i] ? has.replied [i]
    ALT
      time ? AFTER time.now PLUS time.out
        SEQ
          write.full.string (To.screen, "Coordinator timed out")
          write.text.line (To.screen,"Abort decision made")
          coord.state := aborted
          write.full.string (To.screen, "coord state -")
          send.state (To.screen, coord.state)
```

```
                -----------------------------------------------------------------------
-----
                -- Only one of the participants is assumed to be unable to communicate
                   -- Need to abort coordinator, abort all participants that replied,
absorb all
                -- late communications
                  -- The participant that didn't reply will abort itself when it can't
send
                -- its answer message
                -----------------------------------------------------------------------
-----
                PAR
                 IF
                    has.replied [p0] = cam
                      SEQ
                        --- participant is in ready state
                        Out.to.p0 ! acm
                    has.replied [p0] = not.replied
                      SEQ
                        -- p0 not replied => late
                        -- shouldn't happen in this simulation
                        SKIP
                    has.replied [p0] = aam
                      SEQ
                        -- p0 already aborted
                        SKIP
                    TRUE
                      SKIP
                  IF
                    has.replied [p1] = cam
                      SEQ
                        --- participant is in ready state
                        SEQ
                          mess := acm
                          get.next.tick (next.tick, allowed.out.time)
                               OutputOrFail.t (Out.to.p1, mess.array, time, next.tick,
Link.error)
                          IF
                            Link.error
                              SEQ
                                write.full.string (To.screen, "error sending acm to p1")
                                write.full.string (To.screen, " !BLOCKING! ")
                                SKIP
                            NOT Link.error
                              SEQ
                                write.full.string (To.screen, "acm sent ok to p1")
                                SKIP
                    has.replied [p1] = not.replied
                      SEQ
                        -- p1 not replied => link failure
                        write.full.string (To.screen, "p1 not replied")
                        Duff.link.p1 ! error
                    TRUE
                      SKIP

                -----------------------------------------------------------------------
----
                -- synchronous communications need to wait for late comms
                -----------------------------------------------------------------------
---
                decision.made ? vote
            decision.made ? vote
              SEQ
                IF
                 vote = abort
                    SEQ
                      write.text.line (To.screen,"Abort decision made")
                      -----------------------------------------------------------------
-----------
```

```
                      -- abort coordinator and abort all the participants that replied
with cam
                      -- the participants that replied aam should already have aborted
already
                     --------------------------------------------------------------------
-----------
                     coord.state := aborted
                     write.full.string (To.screen, "coord state -")
                     send.state (To.screen, coord.state)
                     TIMER time :
                     VAL delay IS 6 * 15625 :
                     INT time.start :
                     SEQ
                       time ? time.start
                       time ? AFTER time.start PLUS delay
                     PAR
                       IF
                         has.replied[0] = cam
                           SEQ
                             -- commented out to simulate late process
                             Out.to.p0 ! acm
                         TRUE
                           SKIP
                       IF
                         has.replied[1] = cam
                           SEQ
                             mess := acm
                             get.next.tick (next.tick, allowed.out.time)
                                OutputOrFail.t (Out.to.pl, mess.array, time, next.tick,
Link.error)
                             IF
                               Link.error
                                 SEQ
                                    write.full.string (To.screen, "error sending acm to
pl")
                                   write.full.string (To.screen, " !BLOCKING! ")
                                   SKIP
                               NOT Link.error
                                 SEQ
                                   write.full.string (To.screen, "acm sent ok to pl")
                                   SKIP
                         TRUE
                           SKIP
                 vote = commit
                   SEQ
                     -- is coordinator willing ?
                     TIMER time :
                     VAL delay IS 6 * 15625 :
                     INT time.start :
                     SEQ
                       time ? time.start
                       time ? AFTER time.start PLUS delay
                     IF
                       coord.willing
                         SEQ
                           write.text.line (To.screen,"Commit decision made")
                           write.full.string (To.screen, "coord state -")
                           send.state (To.screen, coord.state)
                           TIMER time :
                           VAL delay IS 6 * 15625 :
                           INT time.start :
                           SEQ
                             time ? time.start
                             time ? AFTER time.start PLUS delay
                           PAR  -- commit participants
                             Out.to.p0 ! ccm
                             BOOL sent :
                             SEQ
                               mess := ccm
```

```
                              sent := FALSE
                              WHILE (NOT sent)
                                SEQ
                                  write.char (To.screen, '.')
                                  Reinitialise (Out.to.p1)
                                  get.next.tick (next.tick, allowed.out.time)
                                         OutputOrFail.t (Out.to.p1, mess.array, time,
next.tick, Link.error)
                                  sent := NOT Link.error
                                  write.char (To.screen, ':')
                          coord.state := committed
                      NOT coord.willing
                        SEQ
                          write.text.line (To.screen,"Coord not willing")
                          write.text.line (To.screen,"Abort decision made")
                          coord.state := aborted
                          write.full.string (To.screen, "coord state -")
                          send.state (To.screen, coord.state)
                          TIMER time :
                          VAL delay IS 6 * 15625 :
                          INT time.start :
                          SEQ
                            time ? time.start
                            time ? AFTER time.start PLUS delay
                          PAR
                            -- comment out to simulate late message
                            Out.to.p0 ! acm
                            SEQ
                              mess := acm
                              get.next.tick (next.tick, allowed.out.time)
                               OutputOrFail.t (Out.to.p1, mess.array, time, next.tick,
Link.error)
                              IF
                                Link.error
                                  SEQ
                                      write.full.string (To.screen, "error sending acm
to p1")
                                    write.full.string (To.screen, " !BLOCKING! ")
                                    SKIP
                                NOT Link.error
                                  SEQ
                                    write.full.string (To.screen, "acm sent ok to p1")
                                    SKIP

      write.full.string (To.screen, "coord state -")
      send.state (To.screen, coord.state)
      write.text.line (To.screen, "decision finished")
      stop.mux ! 10 -- any int will do
  :

  SEQ
    INT start :
    SEQ
      From.debug ? start
    PAR
      collect.answers (From.p0, Link.from.p1, part.answered, vote,
                       p1.disconnected, send.to.screen [0])
      decision (part.answered, vote, To.p0, Link.to.p1, stop.screen.mux,
                p1.disconnected, send.to.screen [1])
      multiplexor (send.to.screen, To.debug, stop.screen.mux)
:

PROC participant0 (CHAN OF ANY output, input, Debug.out, Debug.in)
  #USE "2pclib.tsr"
  #USE "\tdsiolib\userio.tsr"
  VAL tt.endstream IS BYTE 24 :
  INT partic.state :
  INT message :
  BOOL partic.willing :
```

```
BOOL partic.late :
BOOL going :
TIMER time :
INT time.now :
INT timed.out :
VAL ticks.per.sec IS 15625 :
VAL allowed.delay IS 10 * ticks.per.sec :  -- in seconds
VAL slow.process IS 2 * ticks.per.sec :
PROC send.state (CHAN OF ANY sink, VAL INT state)
  IF
    state = initial
      write.full.string (sink, " Initial ")
    state = ready
      write.full.string (sink, " Ready ")
    state = undecided
      write.full.string (sink, " Undecided ")
    state = aborted
      write.full.string (sink, " Aborted ")
    state = committed
      write.full.string (sink, " committed ")
:
SEQ
  INT start :
  SEQ
    Debug.in ? start
  partic.willing := TRUE
  partic.late := FALSE
  partic.state := initial
  write.full.string (Debug.out, "partic state is - ")
  send.state (Debug.out, partic.state)
  -- wait for prepare message
  input ? message
  IF
    message = pm
      SEQ
        --write.text.line (Debug.out, "pm got by p0")
        IF
          partic.late
            SEQ
              time ? time.now
              time ? AFTER time.now PLUS slow.process
          NOT partic.late
            SKIP
        -- decide if P1 is willing to commit or not
        IF
          partic.willing
            SEQ
              output ! cam
              partic.state := ready
              write.full.string (Debug.out, "partic state is - ")
              send.state (Debug.out, partic.state)
              input ? message
              IF
                message = ccm
                  SEQ
                    -- all processes are able to commit
                    -- commit this one
                    partic.state := committed
                message = acm
                  SEQ
                    -- some process is unable to commit
                    -- therefore abort this one
                    partic.state := aborted
              time ? time.now
              ALT
                input ? message
                  IF
                    message = ccm
                      SEQ
```

```
                          -- all processes are able to commit
                          -- commit this one
                          partic.state := committed
                      message = acm
                        SEQ
                          -- some process is unable to commit
                          -- therefore abort this one
                          partic.state := aborted
                  time ? AFTER time.now PLUS allowed.delay
                    SEQ
                      write.text.line (Debug.out, "p0 timed out")
                      partic.state := aborted
            NOT partic.willing
              SEQ
                -- unilateral abortion
                partic.state := aborted
                output ! aam
      TRUE
        SKIP
    write.full.string (Debug.out, "partic state is - ")
    send.state (Debug.out, partic.state)
    -- terminate fan.out on mux
    write.text.line (Debug.out, "stop fan")
    Debug.out ! tt.endstream
:

PROC participant1 (CHAN OF ANY Link.to.coord, Link.from.coord,
                                   Debug.out, Debug.in)
  #USE "2pclib.tsr"
  #USE "\tdsiolib\userio.tsr"
  #USE "\tds2lib\reinit.tsr"
  CHAN OF ANY From.coord, To.coord :
  [3]CHAN OF ANY send.to.screen :
  CHAN OF INT stop.mux :
  INT error.source :
  VAL tt.endstream IS BYTE 24 :
  INT partic.state :
  INT message :
  BOOL partic.willing :
  BOOL partic.late :
  BOOL going :
  []BYTE mess.array RETYPES message :
  TIMER time :
  VAL allowed.in.time IS 10*15625 :
  VAL allowed.out.time IS 1*15625 :
  INT next.tick :
  BOOL Link.error :
  INT time.now :
  INT timed.out :
  VAL ticks.per.sec IS 15625 :
  VAL delay IS 5 * ticks.per.sec :  -- in seconds
  VAL slow.process IS 2 * ticks.per.sec :
  PROC send.state (CHAN OF ANY sink, VAL INT state)
    IF
      state = initial
        write.full.string (sink, " Initial ")
      state = ready
        write.full.string (sink, " Ready ")
      state = undecided
        write.full.string (sink, " Undecided ")
      state = aborted
        write.full.string (sink, " Aborted ")
      state = committed
        write.full.string (sink, " committed ")
  :
  PROC get.next.tick (INT next.tick, VAL INT delta)
    TIMER time :
    SEQ
      time ? next.tick
      next.tick := next.tick PLUS delta
  :

  SEQ
    INT start :
    SEQ
      Debug.in ? start
    write.full.string (Debug.out, "p1 started")
    partic.willing := TRUE
    partic.late := FALSE
    partic.state := initial
    write.full.string (Debug.out, "p1 state is - ")
    send.state (Debug.out, partic.state)
    get.next.tick (next.tick, allowed.in.time)
    InputOrFail.t (Link.from.coord, mess.array, time, next.tick, Link.error)
    IF
      Link.error
        SEQ
          write.full.string (Debug.out, "pm not received")
          partic.state := aborted
          SKIP
      NOT Link.error
        SEQ
          IF
            message = pm
              SEQ
                write.full.string (Debug.out, "pm got by p1")
                IF
                  partic.late
                    SEQ
                      time ? time.now
                      time ? AFTER time.now PLUS slow.process
                  NOT partic.late
                    SKIP
                -- decide if P1 is willing to commit or not
                IF
                  partic.willing
                    SEQ
                      SEQ
                        time ? time.now
                        time ? AFTER time.now PLUS delay
                      message := cam
                      get.next.tick (next.tick, allowed.out.time)
                          OutputOrFail.t (Link.to.coord, mess.array, time, next.tick,
Link.error)
                      IF
                        Link.error
                          SEQ
                            write.full.string (Debug.out, "error sending cam")
                            partic.state := aborted
                            SKIP
                        NOT Link.error
                          SEQ
                            write.full.string (Debug.out, "cam sent ok")
                            partic.state := ready
                            write.full.string (Debug.out, "partic state is - ")
                            send.state (Debug.out, partic.state)
                            write.full.string (Debug.out, " BLOCKING ")
                            BOOL got :
                            SEQ
                              got := FALSE
                              WHILE (NOT got)
                                SEQ
                                  Reinitialise (Link.from.coord)
                                  get.next.tick (next.tick, allowed.in.time)
                                     InputOrFail.t (Link.from.coord, mess.array, time,
next.tick, Link.error)
                                  got := NOT Link.error
                            write.full.string (Debug.out, "Decison got ok")
```

```
                              IF
                                message = ccm
                                  SEQ
                                    -- all processes are able to commit
                                    -- commit this one
                                    partic.state := committed
                                message = acm
                                  SEQ
                                    -- some process is unable to commit
                                    -- therefore abort this one
                                    partic.state := aborted
                                message = error
                                  SEQ
                                    -- error detected before getting vote
                                    -- abort participant and terminate
                                    write.full.string (Debug.out, "ERROR in decision")
                                    partic.state := aborted
                    NOT partic.willing
                      SEQ
                        ----------------------------------------------------
                        -- participant unilaterally aborts
                        ----------------------------------------------------
                        message := aam
                        get.next.tick (next.tick, allowed.out.time)
                            OutputOrFail.t (Link.to.coord, mess.array, time, next.tick,
Link.error)
                        IF
                          Link.error
                            SEQ
                              -- aam message not sent
                              write.full.string (Debug.out, "error sending aam")
                              partic.state := aborted
                              SKIP
                          NOT Link.error
                            SEQ
                              write.full.string (Debug.out, "aam sent ok")
                              partic.state := aborted
              TRUE
                SEQ
                  write.full.string (Debug.out, "pm message error")
                  partic.state := aborted
                  SKIP
      write.full.string (Debug.out, "partic state is - ")
      send.state (Debug.out, partic.state)
      write.full.string (Debug.out, "stop fan")
      -- terminate fan.out on mux
      Debug.out ! tt.endstream
:
```

```
PROC multiplexor (CHAN OF ANY From.p0, To.p0, From.p1, To.p1,
                              From.coord, To.coord, From.host, To.host)
  #USE "\tdsiolib\interf.tsr"

  #USE "\tdsiolib\userhdr.tsr"
  #USE "\tdsiolib\filerhdr.tsr"
  #USE "\tdsiolib\slice.tsr"
  PROC fan.out (CHAN OF ANY scrn, screen.out1, screen.out2, terminated)
  -- sends copies of everything received on its input channel
  -- to both of the output channels.
    BYTE tt.bcom:
    BOOL going:
    SEQ
      going := TRUE
      WHILE going
        SEQ
          scrn ? tt.bcom
          IF
            tt.bcom = tt.out.string
              [max.string.size]BYTE str :
```

```
            INT len:
            SEQ
              input.len.bslice(scrn, len, str)
              screen.out1 ! tt.out.string
              output.len.bslice(screen.out1, len, str)
              screen.out2 ! tt.out.string
              output.len.bslice(screen.out2, len, str)
          (tt.bcom = tt.out.byte) OR
           (tt.bcom = tt.ins.char)
            BYTE ch :
            SEQ
              scrn ? ch
              screen.out1 ! tt.bcom; ch
              screen.out2 ! tt.bcom; ch
          tt.bcom = tt.out.int
            INT x :
            SEQ
              scrn ? x
              screen.out1 ! tt.out.int; x
              screen.out2 ! tt.out.int; x
          tt.bcom = tt.goto
            INT x,y:
            SEQ
              scrn ? x;y
              screen.out1 ! tt.goto; x; y
              screen.out2 ! tt.goto; x; y
          tt.bcom = tt.endstream
            SEQ
              screen.out1 ! tt.out.byte;'!';tt.bcom
              screen.out2 ! tt.bcom
              terminated ! 10 -- any will do
              going := FALSE
          TRUE
            SEQ
              screen.out1 ! tt.bcom
              screen.out2 ! tt.bcom
:

VAL link0in IS 4 :
VAL link0out IS 0 :
VAL link1in IS 5 :
VAL link1out IS 1 :
VAL link2in IS 6 :
VAL link2out IS 2 :
VAL link3in IS 7 :
VAL link3out IS 3 :
PLACE From.coord AT link0in:
PLACE To.coord AT link0out:
PLACE From.p0 AT link3in:
PLACE To.p0 AT link3out:
PLACE From.p1 AT link1in:
PLACE To.p1 AT link1out:
PLACE From.host AT link2in:
PLACE To.host AT link2out:
-- channels for scrstream.sink
[3]CHAN OF ANY echo :
-- channels for multiplexor
[3] CHAN OF ANY Send.to.screen :
-- channels for termination
CHAN OF INT Fan.link0, Fan.link1, Fan.link3 :
CHAN OF INT Stop.mux :
#USE "\tdsiolib\userhdr.tsr"
#USE "\tdsiolib\slice.tsr"
#USE "\tdsiolib\filerhdr.tsr"
PROC multiplexor ([]CHAN OF ANY screen.in, CHAN OF ANY screen.out,
                        CHAN OF INT stopper)
-- multiplexes a collection of scrstream output channels onto a
-- single such channel.
-- Each change of input directs output to a new line on the screen.
-- Any input on stopper terminates the multiplexer.
  VAL n IS SIZE screen.in:
  BOOL going :
  INT tt.char :
  BYTE tt.bcom :
  INT any :
  INT current.source :
  PROC check.current.source(INT prev, VAL INT new)
  -- sends *c*n to screen.out if new<>prev, resets prev to be new
    SEQ
      IF
        new = prev
          SKIP
        TRUE
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.out.byte; BYTE ((INT'0') + new)
            screen.out ! tt.out.byte; '>'
            prev := new
  :
  SEQ
    going := TRUE
    current.source := -1
    WHILE going
      PRI ALT
        stopper ? any
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.endstream
            going := FALSE
        ALT i = 0 FOR n
          screen.in[i] ? tt.bcom
            IF
              tt.bcom = tt.out.string
                [max.string.size]BYTE str :
                INT len:
                SEQ
                  input.len.bslice (screen.in[i], len, str)
                  check.current.source(current.source, i)
                  screen.out ! tt.out.string
                  output.len.bslice (screen.out, len, str)
              (tt.bcom = tt.out.byte) OR
               (tt.bcom = tt.ins.char)
                BYTE ch :
                SEQ
                  screen.in[i] ? ch
                  check.current.source(current.source, i)
                  screen.out ! tt.bcom; ch
              tt.bcom = tt.out.int
                INT x :
                SEQ
                  screen.in[i] ? x
                  check.current.source(current.source, i)
                  screen.out ! tt.out.int; x
              tt.bcom = tt.goto
                INT x,y:
                SEQ
                  screen.in[i] ? x;y
                  check.current.source(current.source, i)
                  screen.out ! tt.goto; x; y
              tt.bcom = tt.endstream
                SEQ
                  SKIP
              TRUE
                screen.out ! tt.bcom
:
PROC terminator (CHAN OF INT Fan0, Fan1, Fan3, terminated)
  [3] INT stopped :
  SEQ
```

```
      PAR
        Fan0 ? stopped [0]
        Fan1 ? stopped [1]
        Fan3 ? stopped [2]
      terminated ! 10 -- any int will do
  :

  SEQ
    INT start :
    VAL go IS 10 :  -- any int will do
    SEQ
      From.host ? start
      PAR
        To.p0 ! go
        To.p1 ! go
        To.coord ! go
    PAR
      fan.out (From.p0, Send.to.screen [0], echo [0], Fan.link0)
      scrstream.sink (echo [0])
      fan.out (From.p1, Send.to.screen [1], echo [1], Fan.link1)
      scrstream.sink (echo [1])
      fan.out (From.coord, Send.to.screen [2], echo [2], Fan.link3)
      scrstream.sink (echo [2])
      multiplexor (Send.to.screen, To.host, Stop.mux)
      terminator (Fan.link0, Fan.link1, Fan.link3, Stop.mux)
:

PLACED PAR
   PROCESSOR 0 T8
    PLACE Debug.from.p0 AT link3in :
    PLACE Debug.to.p0 AT link3out :
    PLACE Debug.from.p1 AT link1in :
    PLACE Debug.to.p1 AT link1out :
    PLACE Debug.from.coord AT link0in :
    PLACE Debug.to.coord AT link0out :
    PLACE Debug.from.host AT link2in :
    PLACE Debug.to.host AT link2out :
      multiplexor (Debug.from.p0, Debug.to.p0, Debug.from.p1, Debug.to.p1,
                       Debug.from.coord, Debug.to.coord,Debug.from.host,
                  Debug.to.host)
   PROCESSOR 1 T8
    PLACE Coord.from.p0 AT link2out :
    PLACE Coord.to.p0 AT link2in :
    PLACE Debug.from.p0 AT link3out :
    PLACE Debug.to.p0 AT link3in :
          participant0   (Coord.from.p0,   Coord.to.p0,   Debug.from.p0,
Debug.to.p0)
   PROCESSOR 2 T8
    PLACE Coord.from.p0 AT link1in :
    PLACE Coord.to.p0 AT link1out :
    PLACE Coord.from.p1 AT link3in :
    PLACE Coord.to.p1 AT link3out :
    PLACE Debug.from.coord AT link0out :
    PLACE Debug.to.coord AT link0in :
      coordinator   (Coord.from.p0,  Coord.to.p0,  Coord.from.p1,
                        Coord.to.p1,Debug.from.coord,  Debug.to.coord)
   PROCESSOR 3 T8
    PLACE Coord.from.p1 AT link3out :
    PLACE Coord.to.p1 AT link3in :
    PLACE Debug.from.p1 AT link2out :
    PLACE Debug.to.p1 AT link2in :
          participant1   (Coord.from.p1,   Coord.to.p1,   Debug.from.p1,
Debug.to.p1)
```

```
---------------------------------------------------------------------------
-- This protocol is the extended 2 phase protocol from the paper by Skeen and
-- Stonebraker (IEEE s/w eng. SE-9 No.3 1983) with synchronous comms
--
-- It is implemented here with 1 coordinator and 1 participant
-- Timeouts are added so that the link between the coordinator and the
-- participant can be pulled
-- A centralized protocol is assumed
--
-- Site and link failures are investigated
--
-- see notes 1/3/90 for configuration and 6/3/90
--
-- Results :-
-- (1) Can be made non- blocking and consistent with respect to the link
-- failure
--
---------------------------------------------------------------------------

------------------------------------------------------------------------------
--
-- An extended 2 phase commit protocol runs on the network
--
-- The network is configured as a tree of 1 coordinator 1 participant and
-- a multiplexor process
-- The multiplexor collects debug information from the coordinator and
-- participants and sends it to the host via link 2
--
-- The multiplexor runs on slot 3
-- The coordinator runs on slot 1
-- Participant 1 runs on slot 2
------------------------------------------------------------------------------

-----------------------------------------------------------
-- Channels to host transputer
-----------------------------------------------------------
CHAN OF ANY Debug.from.host, Debug.to.host :

-----------------------------------------------------------
-- Debug channels
-----------------------------------------------------------
CHAN OF ANY Debug.from.p1, Debug.to.p1 :
CHAN OF ANY Debug.from.coord, Debug.to.coord :

-----------------------------------------------------------
-- Network channels between coordinator and participant
-----------------------------------------------------------
CHAN OF ANY Coord.to.p1, Coord.from.p1 :

VAL link0in IS 4 :
VAL link0out IS 0 :
VAL link1in IS 5 :
VAL link1out IS 1 :
VAL link2in IS 6 :
VAL link2out IS 2 :
VAL link3in IS 7 :
VAL link3out IS 3 :

PROC coordinator (CHAN OF ANY From.p1, To.p1,
                              To.debug, From.debug)
  ----------------------------------------------------------------------------
  -- This procedure communicates with participant 1 and also sends debug
  -- info to the host transputer via the multiplexor
  --
  -- The link between the coordinator and participant 1 is the one that can
  -- be removed to demonstrate the resiliency
  ----------------------------------------------------------------------------
  #USE "\tdsiolib\userio.tsr"
  #USE "\tdsiolib\interf.tsr"
```

```
#USE "3pclib.tsr"

VAL  Coord.willing IS FALSE :
[no.of.participants]TIMER timer :
[no.of.participants]INT time.now :
VAL allowed.input.time IS 5 * 15625 :
VAL allowed.output.time IS 3 * 15625 :
[no.of.participants]BOOL Link.error :
INT coord.state, decision :
INT mess.p0, mess.p1 :
[]BYTE mess.array.p0 RETYPES mess.p0 :
[]BYTE mess.array.p1 RETYPES mess.p1 :
[no.of.participants]INT next.tick :
VAL tt.endstream IS BYTE 24 :
PROC send.state (CHAN OF ANY sink, VAL INT state)
  IF
    state = initial
      write.full.string (sink, " Initial ")
    state = ready
      write.full.string (sink, " Ready ")
    state = undecided
      write.full.string (sink, " Undecided ")
    state = aborted
      write.full.string (sink, " Aborted ")
    state = committed
      write.full.string (sink, " committed ")
    state = prepared
      write.full.string (sink, " prepared ")
    TRUE
      write.full.string (sink, " !!ERROR!! ")
:
PROC get.next.tick (INT next.tick, VAL INT delta, TIMER time)
  TIMER time :
  SEQ
    time ? next.tick
    next.tick := next.tick PLUS delta
:

SEQ
  INT start :
  SEQ
    From.debug ? start
  -- initial state
  -- send prepare message to all participants
  coord.state := initial
  write.full.string (To.debug, "coord state -")
  send.state (To.debug, coord.state)
  decision := unknown
  SEQ i = p0 FOR no.of.participants
    Link.error[i] := FALSE
  PAR
    SEQ
      mess.p1 := pm
      get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
      OutputOrFail.t (To.p1, mess.array.p1, timer[p1], next.tick[p1],
                      Link.error[p1])
  IF
    Link.error[p1]
      SEQ
        write.full.string (To.debug, "error sending pm to p1")
        -- assume the link is duff
        -- abort decision made
        coord.state := aborted
    NOT Link.error[p1]
      SEQ
        write.text.line (To.debug, "pm sent ok to p1")
        coord.state := ready
        write.full.string (To.debug, "coord state :-")
        send.state (to.debug, coord.state)
        SEQ i = p0 FOR no.of.participants
          Link.error[i] := FALSE
        get.next.tick (next.tick[p1], allowed.input.time, timer[p1])
        InputOrFail.t (From.p1, mess.array.p1, timer[p1], next.tick[p1],
                       Link.error[p1])
        IF
          Link.error[p1]
            SEQ
              write.full.string (To.debug, "P1 not replied")
              coord.state := aborted
          NOT Link.error[p1]
            write.text.line (To.debug, "Delay - put link if necessary")
            TIMER delay :
            INT now :
            SEQ
              delay ? now
              delay ? AFTER now PLUS (3 *15625)
            SEQ i = p0 FOR no.of.participants
              Link.error[i] := FALSE
            IF
              (mess.p1 = cam) AND (Coord.willing)
                SEQ
                  PAR
                    SEQ
                      mess.p1 := ccm
                      get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
                      OutputOrFail.t (To.p1, mess.array.p1, timer[p1], ext.tick[p1],
                                      Link.error[p1])
                  IF
                    Link.error[p1]
                      SEQ
                        write.textline (To.debug, "error sending ccm to p1")
                        coord.state := aborted
                    NOT Link.error[p1]
                      SEQ
                        write.text.line (To.debug, "P1 got ccm ")
                        coord.state := prepared
                        write.full.string (To.debug, "coord state :-")
                        send.state (to.debug, coord.state)
                        get.next.tick (next.tick[p1], allowed.input.time, timer[p1])
                                InputOrFail.t (From.p1, mess.array.p1, timer[p1],
next.tick[p1],
                                      Link.error[p1])
                        IF
                          Link.error[p1]
                            SEQ
                              write.text.line (To.debug, "ack not received")
                              coord.state := committed
                          NOT Link.error[p1]
                            SEQ
                              write.text.line (To.debug, "ack received ok")
                              coord.state := committed
              (mess.p1 = cam) AND NOT(Coord.willing)
                SEQ
                  PAR
                    SEQ
                      mess.p1 := acm
                      get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
                                OutputOrFail.t (To.p1, mess.array.p1, timer[p1],
next.tick[p1],
                                     Link.error[p1])
                  IF
                    Link.error[p1]
                      SEQ
                        write.text.line (To.debug, "error sending acm to p1")
                        coord.state := aborted
                    NOT Link.error[p1]
                      SEQ
                        write.text.line (To.debug, "P1 got acm ")
```

```
                         coord.state := aborted
                         write.full.string (To.debug, "coord state :-")
                         send.state (to.debug, coord.state)

                TRUE -- abort decision
                  SEQ
                    write.text.line (To.debug, "aam received")
                    coord.state := aborted
    IF
      decision = commit
        write.full.string (To.debug, "Decision = commit")
      decision = abort
        write.full.string (To.debug, "Decision = abort")
    write.full.string (To.debug, "coord state -")
    send.state (To.debug, coord.state)
    write.text.line (To.debug, "decision finished")
    To.debug ! tt.endstream
:
```

```
PROC participant1 (CHAN OF ANY To.coord, From.coord, To.debug,
                         From.debug)
  #USE "3pclib.tsr"
  #USE "\tdsiolib\userio.tsr"
  #USE "\tds2lib\reinit.tsr"    -- extraordinary library

  VAL partic.willing IS TRUE :

  TIMER time :
  INT time.now :
  INT timed.out :
  VAL ticks.per.sec IS 15625 :
  VAL allowed.input.delay IS 10 * ticks.per.sec :  -- in seconds
  [no.of.participants]TIMER timer :
  [no.of.participants]INT time.now :
  VAL allowed.input.time IS 5 * 15625 :
  VAL allowed.output.time IS 3 * 15625 :
  [no.of.participants]BOOL Link.error :
  INT mess.p0, mess.p1 :
  []BYTE mess.array.p0 RETYPES mess.p0 :
  []BYTE mess.array.p1 RETYPES mess.p1 :
  [no.of.participants]INT next.tick :
  VAL tt.endstream IS BYTE 24 :
  INT partic.state :
  INT message :
  PROC get.next.tick (INT next.tick, VAL INT delta, TIMER time)
    TIMER time :
    SEQ
      time ? next.tick
      next.tick := next.tick PLUS delta
  :
  PROC send.state (CHAN OF ANY sink, VAL INT state)
    IF
      state = initial
        write.full.string (sink, " Initial ")
      state = ready
        write.full.string (sink, " Ready ")
      state = undecided
        write.full.string (sink, " Undecided ")
      state = aborted
        write.full.string (sink, " Aborted ")
      state = committed
        write.full.string (sink, " committed ")
      state = prepared
        write.full.string (sink, " prepared ")
      TRUE
        write.full.string (sink, "error state")
  :

  SEQ
```

```
   INT start :
   SEQ
     From.debug ? start
   partic.state := initial
   write.full.string (To.debug, "partic0 state is - ")
   send.state (To.debug, partic.state)
   IF
     partic.willing
       SEQ
         write.full.string (To.debug, "p1 willing")
         -- wait for prepare message
         time ? time.now
         ALT
           time ? AFTER time.now PLUS allowed.input.delay
             SEQ
               -- timed out waiting for pm message
               write.text.line (To.debug, "timed out waiting for pm")
               partic.state := aborted
           From.coord ? message
             IF
               message = pm
                 SEQ
                   write.full.string (To.debug, "Pm received ok")
                   write.full.string (To.debug, "partic0 state is - ")
                   send.state (To.debug, partic.state)
                   SEQ i = p0 FOR no.of.participants
                     Link.error[i] := FALSE
                   SEQ
                     mess.p1 := cam
                     get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
                               OutputOrFail.t  (To.coord, mess.array.p1, timer[p1],
next.tick[p1],
                                     Link.error[p1])
                   IF
                     Link.error[p1]
                       SEQ
                         write.text.line (To.debug, "Failed to send cam")
                         partic.state := aborted
                     NOT Link.error[p1]
                       SEQ
                         write.text.line (To.debug, "cam sent ok")
                         partic.state := ready
                         -- wait for decision
                         time ? time.now
                         ALT
                           time ? AFTER time.now PLUS allowed.input.delay
                             SEQ
                               -- timed out waiting for command
                                   write.text.line (To.debug, "Timed out waiting for
command")
                               partic.state := aborted
                           From.coord ? message
                             SEQ
                                 write.text.line (To.debug, "p1 received decision in
time")
                               IF
                                 message = acm
                                   partic.state := aborted
                                 message = ccm
                                   SEQ i = p0 FOR no.of.participants
                                     Link.error[i] := FALSE
                                   SEQ
                                     mess.p1 := ack
                                                  get.next.tick  (next.tick[p1],
allowed.output.time, timer[p1])
                                          OutputOrFail.t (To.coord, mess.array.p1,
timer[p1], next.tick[p1],
                                                 Link.error[p1])
                                 IF
```

```
                                        Link.error[p1]
                                          SEQ
                                              write.text.line (To.debug, "Failed to send
ack")
                                            partic.state := committed
                                        NOT Link.error[p1]
                                          SEQ
                                            write.text.line (To.debug, "ack sent ok")
                                            partic.state := committed
                                    TRUE
                                      write.full.string (To.debug, "ERROR")

                    TRUE
                      SEQ
                        write.full.string (To.debug, "Error pm not got ")
                        write.int (To.debug, message, 0)
                        SKIP
          NOT partic.willing
            SEQ
              write.full.string (To.debug, "p1 not willing")
              SEQ i = p0 FOR no.of.participants
                Link.error[i] := FALSE
              SEQ
                mess.p1 := aam
                get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
                OutputOrFail.t (To.coord, mess.array.p1, timer[p1], next.tick[p1],
                                Link.error[p1])
              IF
                Link.error[p1]
                  SEQ
                    write.text.line (To.debug, "aam not sent ")
                    partic.state := aborted
                NOT Link.error[p1]
                  SEQ
                    write.text.line (To.debug, "aam sent ok")
                    partic.state := aborted
        write.full.string (To.debug, "partic1 state is - ")
        send.state (To.debug, partic.state)
        write.text.line (To.debug, "stop fan")
        -- terminate fan.out on mux
        To.debug ! tt.endstream
  :
```

**PROC multiplexor (CHAN OF ANY From.p1, To.p1, From.coord,
                         To.coord, From.host, To.host)**

```
  #USE "\tdsiolib\interf.tsr"

  #USE "\tdsiolib\userhdr.tsr"
  #USE "\tdsiolib\filerhdr.tsr"
  #USE "\tdsiolib\slice.tsr"
  PROC fan.out (CHAN OF ANY scrn, screen.out1, screen.out2, terminated)
  -- sends copies of everything received on its input channel
  -- to both of the output channels.
    BYTE tt.bcom:
    BOOL going:
    SEQ
      going := TRUE
      WHILE going
        SEQ
          scrn ? tt.bcom
          IF
            tt.bcom = tt.out.string
              [max.string.size]BYTE str :
              INT len:
              SEQ
                input.len.bslice(scrn, len, str)
                screen.out1 ! tt.out.string
                output.len.bslice(screen.out1, len, str)
                screen.out2 ! tt.out.string
            output.len.bslice(screen.out2, len, str)
        (tt.bcom = tt.out.byte) OR
         (tt.bcom = tt.ins.char)
          BYTE ch :
          SEQ
            scrn ? ch
            screen.out1 ! tt.bcom; ch
            screen.out2 ! tt.bcom; ch
        tt.bcom = tt.out.int
          INT x :
          SEQ
            scrn ? x
            screen.out1 ! tt.out.int; x
            screen.out2 ! tt.out.int; x
        tt.bcom = tt.goto
          INT x,y:
          SEQ
            scrn ? x;y
            screen.out1 ! tt.goto; x; y
            screen.out2 ! tt.goto; x; y
        tt.bcom = tt.endstream
          SEQ
            screen.out1 ! tt.out.byte;'!';tt.bcom
            screen.out2 ! tt.bcom
            terminated ! 10 -- any will do
            going := FALSE
        TRUE
          SEQ
            screen.out1 ! tt.bcom
            screen.out2 ! tt.bcom
:

-- channels for scrstream.sink
[3]CHAN OF ANY echo :
-- channels for multiplexor
[3] CHAN OF ANY Send.to.screen :
-- channels for termination
CHAN OF INT Fan.link1, Fan.link3 :
CHAN OF INT Stop.mux :
#USE "\tdsiolib\userhdr.tsr"
#USE "\tdsiolib\slice.tsr"
#USE "\tdsiolib\filerhdr.tsr"
PROC multiplexor ([]CHAN OF ANY screen.in, CHAN OF ANY screen.out,
                           CHAN OF INT stopper)
-- multiplexes a collection of scrstream output channels onto a
-- single such channel.
-- Each change of input directs output to a new line on the screen.
-- Any input on stopper terminates the multiplexer.
  VAL n IS SIZE screen.in:
  BOOL going :
  INT tt.char :
  BYTE tt.bcom :
  INT any :
  INT current.source :
  PROC check.current.source(INT prev, VAL INT new)
  -- sends *c*n to screen.out if new<>prev, resets prev to be new
    SEQ
      IF
        new = prev
          SKIP
        TRUE
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.out.byte; BYTE ((INT'0') + new)
            screen.out ! tt.out.byte; '>'
            prev := new
  :
  SEQ
    going := TRUE
    current.source := -1
    WHILE going
      PRI ALT
        stopper ? any
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.endstream
            going := FALSE
        ALT i = 0 FOR n
          screen.in[i] ? tt.bcom
            IF
              tt.bcom = tt.out.string
                [max.string.size]BYTE str :
                INT len:
                SEQ
                  input.len.bslice (screen.in[i], len, str)
                  check.current.source(current.source, i)
                  screen.out ! tt.out.string
                  output.len.bslice (screen.out, len, str)
              (tt.bcom = tt.out.byte) OR
               (tt.bcom = tt.ins.char)
                BYTE ch :
                SEQ
                  screen.in[i] ? ch
                  check.current.source(current.source, i)
                  screen.out ! tt.bcom; ch
              tt.bcom = tt.out.int
                INT x :
                SEQ
                  screen.in[i] ? x
                  check.current.source(current.source, i)
                  screen.out ! tt.out.int; x
              tt.bcom = tt.goto
                INT x,y:
                SEQ
                  screen.in[i] ? x;y
                  check.current.source(current.source, i)
                  screen.out ! tt.goto; x; y
              tt.bcom = tt.endstream
                SEQ
                  SKIP
              TRUE
                screen.out ! tt.bcom
:
PROC terminator (CHAN OF INT Fan1, Fan3, terminated)
  [2] INT stopped :
  SEQ
    PAR
      Fan1 ? stopped [0]
      Fan3 ? stopped [1]
    terminated ! 10 -- any int will do
:

SEQ
  INT start :
  VAL go IS 10 :  -- any int will do
  SEQ
    From.host ? start
    PAR
      To.p1 ! go
      To.coord ! go
  PAR
    fan.out (From.p1, Send.to.screen [1], echo [1], Fan.link1)
    scrstream.sink (echo [1])
    fan.out (From.coord, Send.to.screen [2], echo [2], Fan.link3)
    scrstream.sink (echo [2])
    multiplexor (Send.to.screen, To.host, Stop.mux)
    terminator (Fan.link1, Fan.link3, Stop.mux)
:
```

```
PLACED  PAR
   PROCESSOR  0  T8
    PLACE Debug.from.p1 AT link1in :
    PLACE Debug.to.p1 AT link1out :
    PLACE Debug.from.coord AT link0in :
    PLACE Debug.to.coord AT link0out :
    PLACE Debug.from.host AT link2in :
    PLACE Debug.to.host AT link2out :
       multiplexor  (Debug.from.p1,  Debug.to.p1,
                         Debug.from.coord,  Debug.to.coord,Debug.from.host,
                     Debug.to.host)
   PROCESSOR  2  T8
    PLACE Coord.from.p1 AT link3in :
    PLACE Coord.to.p1 AT link3out :
    PLACE Debug.from.coord AT link0out :
    PLACE Debug.to.coord AT link0in :
       coordinator    (Coord.from.p1,  Coord.to.p1,  Debug.from.coord,
                      Debug.to.coord)
   PROCESSOR  3  T8
    PLACE Coord.from.p1 AT link3out :
    PLACE Coord.to.p1 AT link3in :
    PLACE Debug.from.p1 AT link2out :
    PLACE Debug.to.p1 AT link2in :
       participant1  (Coord.from.p1,  Coord.to.p1,  Debug.from.p1,
                     Debug.to.p1)
```

```
----------------------------------------------------------------------------
-- This protocol is the Extended 2 phase protocol from the paper by Skeen and
-- Stonebraker (IEEE s/w eng. SE-9 No.3 1983) optimised for synchronous comms
-- if a participant wants to commit then it accepts the pm message
-- if a participant doesn't want to commit then it rejects the pm
-- Timeouts are used instead of an aam message
--
-- It is implemented here with 1 coordinator and 1 participant
-- Timeouts are added so that the link between the coordinator and the
-- participant can be pulled
-- A centralized protocol is assumed
--
-- Site and link failures are investigated
--
-- see notes 1/3/90 for configuration and 6/3/90
--
-- Results :-
-- (1) Can be made non- blocking and consistent with respect to the link
-- failure
--
----------------------------------------------------------------------------

------------------------------------------------------------
-- Channels to host transputer
------------------------------------------------------------
CHAN OF ANY Debug.from.host, Debug.to.host :
------------------------------------------------------------
-- Debug channels
------------------------------------------------------------
CHAN OF ANY Debug.from.p1, Debug.to.p1 :
CHAN OF ANY Debug.from.coord, Debug.to.coord :

------------------------------------------------------------
-- Network channels between coordinator and participant
------------------------------------------------------------
CHAN OF ANY Coord.to.p1, Coord.from.p1 :

VAL link0in IS 4 :
VAL link0out IS 0 :
VAL link1in IS 5 :
VAL link1out IS 1 :
VAL link2in IS 6 :
VAL link2out IS 2 :
VAL link3in IS 7 :
VAL link3out IS 3 :

PROC coordinator (CHAN OF ANY From.p1, To.p1,
                                      To.debug, From.debug)
  ------------------------------------------------------------------------------
  -- This procedure communicates with participant 1 and also sends debug
  -- info to the host transputer via the multiplexor
  --
  -- The link between the coordinator and participant 1 is the one that can
  -- be removed to demonstrate the resiliency
  ------------------------------------------------------------------------------
  #USE "\tdsiolib\userio.tsr"
  #USE "\tdsiolib\interf.tsr"
  #USE "\tds2lib\reinit.tsr"    -- extraordinary library
  #USE "3pclib.tsr"    -- message library
  VAL  Coord.willing IS TRUE :
  [no.of.participants]TIMER timer :
  [no.of.participants]INT time.now :
  VAL allowed.input.time IS 5 * 15625 :
  VAL allowed.output.time IS 3 * 15625 :
  [no.of.participants]BOOL Link.error :
  INT coord.state, decision :
  INT mess.p1 :
  []BYTE mess.array.p1 RETYPES mess.p1 :
  [no.of.participants]INT next.tick :
```

```
VAL tt.endstream IS BYTE 24 :
PROC send.state (CHAN OF ANY sink, VAL INT state)
  IF
    state = initial
      write.full.string (sink, " Initial ")
    state = ready
      write.full.string (sink, " Ready ")
    state = undecided
      write.full.string (sink, " Undecided ")
    state = aborted
      write.full.string (sink, " Aborted ")
    state = committed
      write.full.string (sink, " committed ")
    state = prepared
      write.full.string (sink, " prepared ")
    TRUE
      write.full.string (sink, " !!ERROR!! ")
:
PROC get.next.tick (INT next.tick, VAL INT delta, TIMER time)
  TIMER time :
  SEQ
    time ? next.tick
    next.tick := next.tick PLUS delta
:

SEQ
  INT start :
  SEQ
    From.debug ? start
  -- initial state
  -- send prepare message to all participants
  coord.state := initial
  write.full.string (To.debug, "coord state -")
  send.state (To.debug, coord.state)
  decision := unknown
  SEQ i = p0 FOR no.of.participants
    Link.error[i] := FALSE
  PAR
    SEQ
      mess.p1 := pm
      get.next.tick (next.tick[p1], allowed.output.time, timer[p1])
      OutputOrFail.t (To.p1, mess.array.p1, timer[p1], next.tick[p1],
                      Link.error[p1])
  IF
    Link.error[p1]
      SEQ
        write.full.string (To.debug, "error sending pm to p1")
        -- p1 either not willing to commit or the link is duff
        -- abort decision made, assume replied AAM
        coord.state := aborted
    NOT Link.error[p1]
      SEQ
        write.text.line (To.debug, "pm sent ok to p1")
        coord.state := ready
        write.full.string (To.debug, "coord state -")
        send.state (To.debug, coord.state)
        SEQ i = p0 FOR no.of.participants
          Link.error[i] := FALSE
        get.next.tick (next.tick[p1], allowed.input.time, timer[p1])
        InputOrFail.t (From.p1, mess.array.p1, timer[p1], next.tick[p1],
                       Link.error[p1])
        IF
          Link.error[p1]
            SEQ
              coord.state := aborted
          NOT Link.error[p1]
            SEQ
              write.text.line (To.debug, "Delay - pull link if necessary")
              TIMER delay :
```

```
            INT now :
            SEQ
              delay ? now
              delay ? AFTER now PLUS (3 *15625)
            SEQ i = p0 FOR no.of.participants
              Link.error[i] := FALSE
            IF
              (mess.p1 = cam) AND (Coord.willing)
                SEQ
                  coord.state := prepared
                  write.text.line (To.debug, "Coord.state = prepared")
                  PAR
                    SEQ
                      mess.p1 := ccm
                      get.next.tick (next.tick[p1], allowed.output.time,
timer[p1])

                      OutputOrFail.t (To.p1, mess.array.p1, timer[p1],
next.tick[p1],
                                      Link.error[p1])
                  IF
                    Link.error[p1]
                      SEQ
                        write.text.line (To.debug, "error sending ccm to p1")
                        coord.state := aborted
                    NOT Link.error[p1]
                      SEQ
                        write.text.line (To.debug, "p1 got ccm")
                        coord.state := committed
              TRUE
                SEQ
                  write.text.line (To.debug, "No need to abort p1 ")
                  write.text.line (To.debug, "It will timeout and abort ")
                  coord.state := aborted
            IF
              Link.error[p1]
                SEQ
                  write.text.line (To.debug, "error sending decision to p1")
              NOT Link.error[p1]
                SEQ
                  ----------------------------------------
                  -- Either p1 voted to abort
                  -- or received the decision ok
                  ----------------------------------------
                  write.text.line (To.debug, "Decisions sent ok")
    write.full.string (To.debug, "coord state -")
    send.state (To.debug, coord.state)
    write.text.line (To.debug, "decision finished")
    To.debug ! tt.endstream
:
```

PROC participant1 (CHAN OF ANY To.coord, From.coord, To.debug, From.debug)

```
  ----------------------------------------------------------------------------
  -- Extended 2 phase commit participant : optimised for synchronous comms
  -- i.e. aam and acm messages removed : noe inferred by timeouts
  -- participant doesn't start until user sends 'start' message
  -- messages receieved are : pm, ccm
  -- messages sent are : cam
  -- variable Partic.willing is used to simulate a check on conditions
  --
  ----------------------------------------------------------------------------

  #USE "\tds2lib\reinit.tsr"    -- extraordinary library
  #USE "3pclib.tsr"    -- message library
  #USE "\tdsiolib\userio.tsr"

  VAL partic.willing IS TRUE :
  VAL tt.endstream IS BYTE 24 :
  INT partic.state :
```

```
INT message :
BOOL Link.error :
[]BYTE mess.array RETYPES message :
TIMER time :
INT time.now :
INT timed.out :
INT next.tick :
VAL ticks.per.sec IS 15625 :
VAL allowed.input.delay IS 10 * ticks.per.sec :  -- in seconds
VAL allowed.output.delay IS 5 * ticks.per.sec :
PROC send.state (CHAN OF ANY sink, VAL INT state)
  IF
    state = initial
      write.full.string (sink, " Initial ")
    state = ready
      write.full.string (sink, " Ready ")
    state = undecided
      write.full.string (sink, " Undecided ")
    state = aborted
      write.full.string (sink, " Aborted ")
    state = committed
      write.full.string (sink, " committed ")
    state = prepared
      write.full.string (sink, " prepared ")
    TRUE
      write.full.string (sink, "error state")
:
PROC get.next.tick (INT next.tick, VAL INT delta, TIMER time)
  TIMER time :
  SEQ
    time ? next.tick
    next.tick := next.tick PLUS delta
:

SEQ
  INT start :
  SEQ
    From.debug ? start
  partic.state := initial
  write.full.string (To.debug, "particl state is - ")
  send.state (To.debug, partic.state)
  time ? time.now
  ALT
    time ? AFTER time.now PLUS allowed.input.delay
      partic.state := aborted
    From.coord ? message
      IF
        message = pm
          SEQ
            write.full.string (To.debug, "Pm received ok")
            IF
              partic.willing
                SEQ
                  SEQ
                    write.full.string (To.debug, "p1 sending cam to coord")
                    write.text.line (To.debug, "Delay - pull link if necessary")
                    TIMER delay :
                    INT now :
                    SEQ
                      delay ? now
                      delay ? AFTER now PLUS (3 *15625)
                    message := cam
                    get.next.tick (next.tick, allowed.output.delay, time)
                    OutputOrFail.t (To.coord, mess.array, time, next.tick,
                                    Link.error)
                  IF
                    Link.error
                      SEQ
                        -- failed to send ccm
                      partic.state := aborted
                  NOT Link.error
                    SEQ
                      -- particpant ready to commit
                      partic.state := ready
                      write.full.string (To.debug, "particl state is - ")
                      send.state (To.debug, partic.state)
                      -- wait for commit message
                      time ? time.now
                      ALT
                        time ? AFTER time.now PLUS allowed.input.delay
                          SEQ
                            -- timed out waiting for command
                            write.text.line (To.debug, "Timed out waiting for
command")

                            partic.state := aborted
                        From.coord ? message
                          SEQ
                            write.text.line (To.debug, "p1 received decision in
time")

                            IF
                              message = acm
                                partic.state := aborted
                              message = ccm
                                partic.state := committed
                              TRUE
                                write.full.string (To.debug, "ERROR")
              NOT partic.willing
                SEQ
                  write.full.string (To.debug, "p1 not willing")
                  partic.state := aborted
          TRUE
            SEQ
              write.full.string (To.debug, "Error pm not got ")
              write.int (To.debug, message, 0)
              SKIP
      write.full.string (To.debug, "particl state is - ")
      send.state (To.debug, partic.state)
      write.text.line (To.debug, "stop fan")
      -- terminate fan.out on mux
      To.debug ! tt.endstream
:
```

```
PROC multiplexor (CHAN OF ANY From.p1, To.p1, From.coord,
                      To.coord, From.host, To.host)
  #USE "\tdsiolib\interf.tsr"

  #USE "\tdsiolib\userhdr.tsr"
  #USE "\tdsiolib\filerhdr.tsr"
  #USE "\tdsiolib\slice.tsr"
  PROC fan.out (CHAN OF ANY scrn, screen.out1, screen.out2, terminated)
  -- sends copies of everything received on its input channel
  -- to both of the output channels.
    BYTE tt.bcom:
    BOOL going:
    SEQ
      going := TRUE
      WHILE going
        SEQ
          scrn ? tt.bcom
          IF
            tt.bcom = tt.out.string
              [max.string.size]BYTE str :
              INT len:
              SEQ
                input.len.bslice(scrn, len, str)
                screen.out1 ! tt.out.string
                output.len.bslice(screen.out1, len, str)
                screen.out2 ! tt.out.string
              output.len.bslice(screen.out2, len, str)
          (tt.bcom = tt.out.byte) OR
           (tt.bcom = tt.ins.char)
            BYTE ch :
            SEQ
              scrn ? ch
              screen.out1 ! tt.bcom; ch
              screen.out2 ! tt.bcom; ch
          tt.bcom = tt.out.int
            INT x :
            SEQ
              scrn ? x
              screen.out1 ! tt.out.int; x
              screen.out2 ! tt.out.int; x
          tt.bcom = tt.goto
            INT x,y:
           SEQ
              scrn ? x;y
              screen.out1 ! tt.goto; x; y
              screen.out2 ! tt.goto; x; y
          tt.bcom = tt.endstream
            SEQ
              screen.out1 ! tt.out.byte;'!';tt.bcom
              screen.out2 ! tt.bcom
              terminated ! 10 -- any will do
              going := FALSE
          TRUE
            SEQ
              screen.out1 ! tt.bcom
              screen.out2 ! tt.bcom
:

-- channels for scrstream.sink
[3]CHAN OF ANY echo :
-- channels for multiplexor
[3] CHAN OF ANY Send.to.screen :
-- channels for termination
CHAN OF INT Fan.link1, Fan.link3 :
CHAN OF INT Stop.mux :
#USE "\tdsiolib\userhdr.tsr"
#USE "\tdsiolib\slice.tsr"
#USE "\tdsiolib\filerhdr.tsr"
PROC multiplexor ([]CHAN OF ANY screen.in, CHAN OF ANY screen.out,
                        CHAN OF INT stopper)
-- multiplexes a collection of scrstream output channels onto a
-- single such channel
-- Each change of input directs output to a new line on the screen.
-- Any input on stopper terminates the multiplexer.
  VAL n IS SIZE screen.in:
  BOOL going :
  INT tt.char :
  BYTE tt.bcom :
  INT any :
  INT current.source :
  PROC check.current.source(INT prev, VAL INT new)
  -- sends *c*n to screen.out if new<>prev, resets prev to be new
    SEQ
      IF
        new = prev
          SKIP
        TRUE
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.out.byte; BYTE ((INT'0') + new)
            screen.out ! tt.out.byte; '>'
            prev := new
  :
  SEQ
    going := TRUE
    current.source := -1
    WHILE going
      PRI ALT
        stopper ? any
          SEQ
            screen.out ! tt.out.byte; '*c'; tt.out.byte; '*n'
            screen.out ! tt.endstream
            going := FALSE
        ALT i = 0 FOR n
          screen.in[i] ? tt.bcom
            IF
              tt.bcom = tt.out.string
                [max.string.size]BYTE str :
                INT len:
                SEQ
                  input.len.bslice (screen.in[i], len, str)
                  check.current.source(current.source, i)
                  screen.out ! tt.out.string
                  output.len.bslice (screen.out, len, str)
              (tt.bcom = tt.out.byte) OR
               (tt.bcom = tt.ins.char)
                BYTE ch :
                SEQ
                  screen.in[i] ? ch
                  check.current.source(current.source, i)
                  screen.out ! tt.bcom; ch
              tt.bcom = tt.out.int
                INT x :
                SEQ
                  screen.in[i] ? x
                  check.current.source(current.source, i)
                  screen.out ! tt.out.int; x
              tt.bcom = tt.goto
                INT x,y:
                SEQ
                  screen.in[i] ? x;y
                  check.current.source(current.source, i)
                  screen.out ! tt.goto; x; y
              tt.bcom = tt.endstream
                SEQ
                  SKIP
              TRUE
                screen.out ! tt.bcom
:
PROC terminator (CHAN OF INT Fan1, Fan3, terminated)
  [2] INT stopped :
  SEQ
    PAR
      Fan1 ? stopped [0]
      Fan3 ? stopped [1]
    terminated ! 10 -- any int will do
:

SEQ
  INT start :
  VAL go IS 10 :  -- any int will do
  SEQ
    From.host ? start
    PAR
      To.p1 ! go
      To.coord ! go
  PAR
    fan.out (From.p1, Send.to.screen [1], echo [1], Fan.link1)
    scrstream.sink (echo [1])
    fan.out (From.coord, Send.to.screen [2], echo [2], Fan.link3)
    scrstream.sink (echo [2])
    multiplexor (Send.to.screen, To.host, Stop.mux)
    terminator (Fan.link1, Fan.link3, Stop.mux)
:
```

```
PLACED  PAR
   PROCESSOR  0  T8
    PLACE Debug.from.p1 AT link1in :
    PLACE Debug.to.p1 AT link1out :
    PLACE Debug.from.coord AT link0in :
    PLACE Debug.to.coord AT link0out :
    PLACE Debug.from.host AT link2in :
    PLACE Debug.to.host AT link2out :
    multiplexor (Debug.from.p1, Debug.to.p1,
                 Debug.from.coord, Debug.to.coord,Debug.from.host, Debug.to.host)
   PROCESSOR  2  T8
    PLACE Coord.from.p1 AT link3in :
    PLACE Coord.to.p1 AT link3out :
    PLACE Debug.from.coord AT link0out :
    PLACE Debug.to.coord AT link0in :
    coordinator  (Coord.from.p1, Coord.to.p1, Debug.from.coord, Debug.to.coord)
   PROCESSOR  3  T8
    PLACE Coord.from.p1 AT link3out :
    PLACE Coord.to.p1 AT link3in :
    PLACE Debug.from.p1 AT link2out :
    PLACE Debug.to.p1 AT link2in :
    participant1 (Coord.from.p1, Coord.to.p1, Debug.from.p1, Debug.to.p1)
```