Preventing State Divergence in Replicated Distributed Systems

by

Alan Tully

Ph.D. Thesis

September 1990

The University of Newcastle upon Tyne

Computing Laboratory

Abstract

N–Modular Redundancy (NMR) is a form of active replication in which each processor is replicated to form a node and each processor replica within the node executes the same set of software component replicas. Communication between nodes, in the form of messages, passes through a voting mechanism by which processor failures are masked. When the degree of replication is three, the technique is known as Triple Modular Redundancy (TMR) and can tolerate the failure of a single node processor.

For voting to be successful, non–faulty software component replicas must output identical messages in an identical order. If we assume that software components are *deterministic*, then we need only ensure that the replicas process identical input messages in an identical order. Such software components conform to the well understood and researched *state machine* model of active replication.

However, most distributed programs employ mechanisms not incorporated in the *state machine* model such as timeouts and prioritized messages. These potential sources of *non–determinism* could lead to a divergence of state among software component replicas which could then produce inconsistent responses to identical input messages, thereby defeating the NMR voting mechanism.

The main contributions of this thesis are:

(i) To present an architecture for active replicated processing which may be applied to any distributed system.

(ii) To present a more expressive, enhanced model for software components which incorporates non–determinism and show how a system of such software components may be replicated, using a single well–defined generic mechanism (the *order process*) to prevent state divergence. Since the problem of identical ordering can be formulated as the *interactive consistency problem* which is solvable in the presence of arbitrary (Byzantine) failures, the approach presented in this thesis, unlike any other published to date, is capable of tolerating such failures.

Acknowledgements

Firstly I would like to thank my supervisor, Professor Santosh Shrivastava, for suggesting this area of research and for reading and commenting upon the numerous drafts of this thesis.

I would also like to thank Dr. Paul Ezhilchelvan, Dr. Neil Speirs, Dr. Stuart Wheater and Mark Little, for their many useful comments.

Finally I would like to thank my family for their support and encouragement, which they gave me during my studies.

Financial support for much of the work described in this thesis was provided by grants from the Science and Engineering Research Council and ESPRIT Project DELTA–4.

Chapter 1 : Introduction

A large class of computer applications specify a performance requirement which far exceeds the capability of any single processor. Even as processors become faster due to new materials, fabrication methods and internal organization, new applications arise which continue to exceed current capabilities. It is for this reason that computer architects have turned to various forms of parallelism to satisfy their needs.

When parallel systems are designed, high performance is often achieved by using application–specific hardware and sacrificing versatility. With the advent of VLSI however, computing hardware costs have been reduced to such a level that large networks of general–purpose processors have become economically viable. A custom–designed system will always be able to out–perform a regular network of general–purpose processors. However, this apparent inefficiency may in many cases be offset by an increase in the number of processors. The reduction in design costs will more than pay for the moderate increase in hardware costs for a broad range of applications.

The construction of massive networks of processors raises the question of reliability. If a single processor failure results in the failure of the system as a whole then that system is of little practical use. Fault–avoidance techniques may be used to minimize the probability of failure but that probability cannot be made negligible in large and complex systems. Additional steps must be taken to ensure continuous system operation by providing tolerance to individual processor failures.

Fault–tolerance techniques which employ redundancy can prevent failure of a system in the presence of faults. *Passive replication* techniques provide for the re–execution of software components on a non–faulty processor. *Active replication* techniques replicate software components and concurrently execute each replica on an

independent processor. In this way, the failure of a bounded number of processors can be *masked* without a further time penalty. The latter solution is particularly attractive when the computing system must provide a *real–time* response to external events.

Some active replication techniques make the assumption that processors have *fail–silent* semantics. That is, a processor either produces the correct response (as specified) or no response at all. A minimum of $m + 1$ software component replicas are necessary to tolerate m processor failures (in the worst case where m failures occur, the outputs of the one remaining non–faulty replica may be trusted). An alternative view assumes that processors have *fail–arbitrary* semantics (*Byzantine* failure [2][11]). That is, the behaviour of a faulty processor is undefined. A minimum of $2m + 1$ replicas are then necessary to tolerate m processor failures (in the worst case where m failures occur, the outputs of the remaining $m + 1$ non–faulty replicas will still form a majority).

N–Modular Redundancy (NMR) is a form of active replication in which each processor is replicated to form a node and each processor replica within the node executes the same set of software component replicas. Communication between nodes, in the form of messages, passes through a voting mechanism by which processor failures are masked. When the degree of replication is three, the technique is known as Triple Modular Redundancy (TMR) and can tolerate the failure of a single node processor.

For voting to be successful, non–faulty software component replicas must output identical messages in an identical order. If we assume that software components are *deterministic*, then we need only ensure that the replicas process identical input messages in an identical order. Such software components conform to the well understood and researched *state machine* model of active replication [7] where, "Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in the system".

However, most distributed programs employ mechanisms not incorporated in the state machine model such as timeouts and prioritized messages. These potential

sources of *non–determinism* could lead to a divergence of state among software component replicas which could then produce inconsistent responses to identical input messages, thereby defeating the NMR voting mechanism. This thesis addresses the problem of preventing state divergence in active replicated processing using an enhanced processing model.

The main contributions of this thesis are:

(i) To present an architecture for active replicated processing which may be applied to any distributed system (reported in [8]).

(ii) To present a more expressive, enhanced model for software components which incorporates non–determinism and show how a system of such software components may be replicated, using a single well–defined generic mechanism (the *order process*) to prevent state divergence (reported in [9]). Since the problem of identical ordering can be formulated as the *interactive consistency problem* which is solvable in the presence of arbitrary (*Byzantine* [2]) failures, the approach presented in this thesis, unlike any other published to date, is capable of tolerating such failures.

The work reported in this thesis forms the basis of a practical architecture that the author and colleagues are implementing for a real–time application (initial work reported in [10]).

Chapter 2 introduces the state machine as a model for active replicated processing and shows how the consistent message ordering requirement of the replicated state machine may be achieved using solutions to the interactive consistency problem. Interactive consistency is examined in more detail using the well–known Byzantine Generals Problem analogy for which several solutions are then presented. Finally, expansion of the state machine model to incorporate non–deterministic mechanisms for real–time systems is considered, to show that consistent message ordering alone is not sufficient to prevent state divergence among process replicas.

Chapter 3 presents the architectures of several existing systems which employ a range of passive and/or active replication techniques to achieve fault-tolerance, with a view to showing how each prevents state divergence.

Chapter 4 describes the proposed architecture of a multi-processor distributed system designed to tolerate Byzantine faults. A processing model is presented then replicated using TMR techniques. A processor interconnection scheme is developed to support the replicated processing model which remains fully connected after one processor failure per TMR node. Finally, a degenerate form of the node is described in which process and processor triads are replaced by process and processor pairs to create a node with *fail-silent* semantics.

Chapter 5 introduces the order process as a mechanism for preventing state divergence in software components with non-deterministic message selection (state machine model) and shows how it may be implemented using protocols which tolerate various classes of faults, up to and including Byzantine faults.

In Chapter 6, the order process is used to prevent state divergence in an enhanced model which allows more complex message selection criteria (*generic input function*), asynchronous external events, non-deterministic processing, semaphores (to allow shared-memory inter-process communication) and real-time clocks.

Chapter 7 shows how the input constructs of several current languages may be mapped onto the generic input function. Because state divergence can be prevented when using the *generic input function*, it can also be prevented when executing the example constructs.

Chapter 8 describes a practical design of the mechanisms presented in this thesis using the *client/server* facilities provided by the Helios operating system.

Finally, Chapter 9 considers the implications of using the techniques presented in this thesis and reflects on possible future enhancements.

Chapter 2 : Understanding Active Replication

2.1 Introduction

In active replication, all the non–faulty replicas must produce identical output messages in an identical order. This requires that non–faulty replicas carry–out identical processing on an identical sequence of identical input messages. The state machine model [7] to be discussed in this chapter provides a rigourous setting for explaining the principles of active replication. As will be shown, to support active replication, protocols for message agreement and order are required. These protocols can be formulated in terms of the interactive consistency problem which is also discussed in this chapter. Finally, limitations of the state machine model which make it less suitable for real–time distributed systems are discussed. The removal of these limitations will introduce problems of non–determinism amongst replicas, which must be addressed for active replication to be useful. Solutions to these problems will be developed in subsequent chapters of this thesis.

2.2 The State Machine

A state machine [7] consists of state variables which implement its state and commands which transform its state and produce output. A command, which is implemented by a deterministic program, is invoked by a request from a client. Multiple clients may make multiple concurrent requests but the commands will be executed strictly sequentially. Thus a state machine will possess the following property:

P1 The outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.

Requests are processed by a state machine in an order consistent with causality:

O1 Multiple requests issued by a single client to a given state machine *sm* are processed in the order they were issued.

O2 If the fact that request *r* was made to *sm* by client *c* could have caused a request *r'* to be made to *sm* by another client *c'*, then *r* is processed before *r'*.

The scenario described by *O2* is possible if the request *r* made to *sm* was followed by a request *r''* made to some other state machine *sm''* which then itself acted as a client *c'* by making the request *r'* to *sm*. Because the making of a request and the receipt of any reply are separated in the state machine model, it is possible for the request *r'* to arrive at *sm* before the request *r*. Messages cannot therefore be processed simply in the order in which they are received at *sm*.

A state machine may be expressed by a process accepting requests in the form of messages and performing the actions they specify as shown in Figure 1.

```
process state_machine
    var m:message
    cycle
        receiveany(m)
        action(m)
    end
```

Figure 1 The State Machine

The message returned by a call to the function receiveany(m) must satisfy *O1* and *O2* and action(m) must satisfy *P1*.

2.3 Replicating the State Machine

A fault-tolerant state machine can be implemented by replicating it and executing each replica on a distinct processor in a distributed system. The outputs of the replicas may then be combined though voting to mask the failure of some of the processors executing state machine replicas. However, if voting is to be successful, it is necessary to ensure that non-faulty replicas produce the same outputs. If each replica starts in the same initial state and processes requests in the same order then because of ***P1***, each will produce the same outputs. Thus a fault-tolerant state machine requires the fault-tolerant implementation of the ***Order*** abstraction:

Order Requests are processed in the same order by every non-faulty state machine replica.

This in turn requires the fault-tolerant implementation of the ***Agreement*** abstraction:

Agreement Every non-faulty copy of the state machine receives every request.

Agreement may be implemented by any protocol which allows the processor executing a client (the transmitter) to send a request (the value) to all the processors executing state machine replicas (the receivers) such that:

IC1 All non-faulty receivers agree on the same value.

IC2 If the transmitter is non-faulty then all non-faulty receivers use its value as the one they agree on.

These are the interactive consistency conditions of the *Byzantine Generals Problem* and may be satisfied using any of the solutions to that problem. Several solutions will be presented in Section 2.4.

Order may be implemented by assigning to each request a unique identifier and ensuring that state machine replicas process requests according to a total ordering relation on these identifiers. Total order is necessary as processing requests in the

order they are received by the replicas does not necessarily satisfy the *Order* abstraction. For example, two requests could be received by one state machine replica in one order while being received by another replica in the other order. Two methods of assigning unique identifiers at the point of issue (the client) will be presented in Section 2.5. Both methods satisfy ***O1*** and ***O2***; the first uses logical clocks while the second uses approximately synchronized real–time clocks.

In Chapter 5, other practical solutions will be presented employing atomic broadcast protocols which simultaneously satisfy *Agreement* and *Order*.

Note that a client may itself be structured as a state machine and therefore it may also be made fault–tolerant through replication.

2.4 Interactive Consistency

2.4.1 *Introduction*

The *Byzantine Generals Problem* [2] has been proposed as an abstract way of expressing the *interactive consistency* problem encountered in distributed computing systems constructed using components which, when they fail, can exhibit arbitrary behaviour. As such, it provides a convenient way of relating the problems and algorithms of this thesis to each other and to the *interactive consistency* problem itself.

A *loyal* general may represent a correctly functioning processor while a *traitorous* general may represent a failed processor. The messages sent between processors are represented by orders sent from a commanding general to his lieutenant generals.

The *Byzantine Generals Problem* is introduced in Section 2.4.2 followed by solutions based on *oral* messages (Section 2.4.3) and *signed* messages (Section 2.4.4). Finally, the main findings of this section are summarized in Section 2.4.5.

2.4.2 *The Byzantine Generals Problem*

A number of divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with each other only via messengers.

Each general g_i observes the enemy and independently forms an opinion v_i on whether the army should *attack* or *retreat*, then communicates v_i to the other generals g_j ($i \neq j$) using messengers. g_i collects the opinions v_j of all the other generals from their messengers to form, along with his own opinion, V_i:

$$V_i = \textit{setof } v_j \textit{ for } j = 1..n \quad \textit{(where } n = \textit{the number of generals)}$$

g_i then performs a majority vote on V_i to reach a decision d_i on whether to *attack* or *retreat*:

$$d_i = \textit{maj } V_i$$

Some of the generals will be loyal while some will be traitorous but the following conditions must always be satisfied:

condA All loyal generals reach the same decision, d_i.

condB A small number of traitorous generals cannot cause the loyal generals to reach a *bad* decision.

A decision to attack may be considered *bad* for example, if a large majority of loyal generals are of the opinion that the army should retreat (because the enemy is too strong). A decision to retreat may be considered *bad*, if a large majority of loyal generals are of the opinion that the army should attack (lest the enemy grow in strength).

A loyal general g_j will always send the same value of v_j to all the other generals. So if all generals are loyal, every general g_i will receive the same set of opinions and will therefore reach the same decision, satisfying ***condA.***:

$$\textbf{for all } i = 1..n \textbf{ and } j = 1..n \quad V_i = V_j \Rightarrow \textbf{maj } V_i = \textbf{maj } V_j \Rightarrow d_i = d_j$$

Now suppose that one general g_f is traitorous, he could send a different view of his opinion v_f to different generals. Some general g_i would then receive a different set of opinions (V_i) to some other general g_j (V_j). If the opinions of the loyal generals were divided equally between *attack* and *retreat*, then they may not reach the same decision, thereby violating ***condA***:

$$V_i \neq V_j \Rightarrow \textbf{maj } V_i \neq \textbf{maj } V_j \Rightarrow d_i \neq d_j$$

Therefore to satisfy ***condA*** in the presence of a traitorous general:

cond1 Every loyal general g_i must receive the same set of opinions V_i.

That is, for every general g_j, whether or not loyal:

cond1' Every loyal general g_i must obtain the same value for v_j.

In satisfying ***condA***, ***condB*** must not be violated. Therefore, for every loyal general g_j:

cond2 Every loyal general g_i must receive v_j uncorrupted.

Since conditions ***cond1'*** and ***cond2*** are both conditions on the opinion v_j of general g_j as received by general g_i, the problem is reduced to ensuring that a general g_j may send his opinion v_j to the other generals. This problem may be re–phrased in terms of a commanding general g_j sending an order (use v_j as my opinion) to his lieutenants. This is known as the *Byzantine Generals Problem*:

Byzantine Generals Problem

A commanding general must send an order to his lieutenant generals such that the following conditions are satisfied:

IC1 All loyal lieutenants obey the same order.

IC2 If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Conditions *IC1* and *IC2* are called the *interactive consistency* conditions, and are identical to the message-based formulation presented in Section 2.3.

Conditions ***condA*** and ***condB*** may then be satisfied by repeatedly applying some solution to the *Byzantine Generals Problem*, with each general in turn acting as the commander.

Two solutions will now be considered; one using *oral* messages and the other using *signed* messages. The reader is referred to [2] for a formal proof of the algorithms.

2.4.3 *A Solution using Oral Messages*

It is assumed that a general may send a message directly to any other general and that messengers are loyal (a traitorous messenger may be modelled by the traitorous general who sent him). So a traitorous general cannot interfere with the communication between any other generals. It is therefore assumed that:

A1 Every message that is sent is delivered correctly.

A2 The receiver of a message knows who sent it.

It is further assumed that:

A3 The absence of a message may be detected.

Satisfying *A3* is not straightforward; a general cannot detect, in finite time, that a message will never arrive. But, if the time taken for the generation and transmission of

a message by a loyal general is bounded to some fixed value Δ, then a general may assume that any message arriving after Δ must be from a traitorous general and may be discarded. A message is therefore assumed to be absent if it does not arrive by T + Δ, where T is the time of transmission. For such timeouts to be meaningful, all generals must know the actual time of transmission T. The value of T may be implicit (eg periodic) or explicitly contained within the message (timestamp). In the later case the general would have to receive at least one copy of the message, either direct or via another general (otherwise he could not determine T). In either case, each general measures time by his own clock, so the clocks of all loyal generals must be synchronized to within some bounded time ε (see Section 5.3).

If a traitorous commanding general does not send an order to one of his lieutenants, the lieutenant assumes *retreat*.

For a solution to the *Byzantine Generals Problem* satisfying *IC1* and *IC2* using oral messages, which tolerates m traitorous generals, there must be at least $3m + 1$ generals in total.

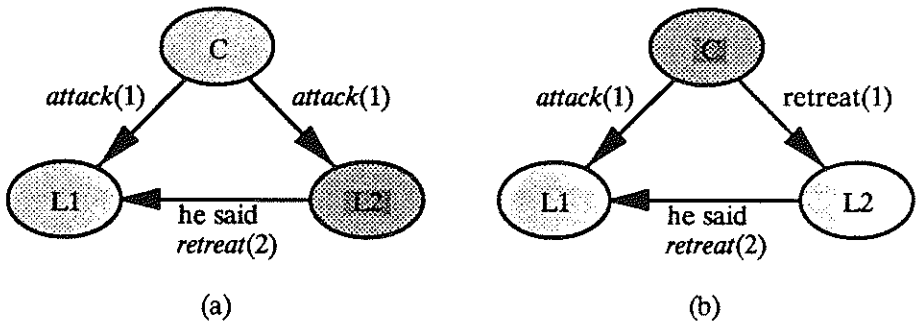

Figure 2 Insufficient Generals for Oral Messages

Consider the converse case ($n < 3m + 1$) from the point of view of lieutenant *L1* as shown in Figure 2, where $n = 3$ and $m = 1$.

Each general executes the appropriate 2-stage algorithm of Figure 3. The timeout clause is necessary to handle a traitorous general who refuses to send (commander) or relay (lieutenant) an order.

In the scenario of Figure 2(a), the commander is loyal and in stage 1 orders *attack*, but the traitorous lieutenant *L2* lies to the loyal lieutenant *L1* in stage 2 by saying that he received the order to *retreat*. In the scenario of Figure 2(b), the commander is traitorous and orders *L1* to *attack* while ordering *L2* to *retreat* in stage 1. In stage 2, *L2*

```
process commander
    var v:order
    cycle
        formulate_order(v)
        send_to_all_lieutenants(v)
    end
end

process lieutenant
    var v, v', vc:order
    cycle
        within t do
            receive_from_commander(v)
        timeout
            v = retreat
        od
        send_to_other_lieutenant(v)
        within t do
            receive_from_other_lieutenant(v')
        timeout
            v' = retreat
        od
        vc = decide_order(v,v')
    end
end
```

Stage 1

Stage 1

Stage 2

Figure 3 Oral Message Algorithms

correctly informs *L1* that he received the order to *retreat*. These two scenarios appear exactly the same to *L1*. Hence he doesn't know the identity of the traitorous general nor what message was sent to *L2*. Hence to satisfy *IC2*, *L1* must obey the order received

directly from the commander in both scenarios, that is to *attack*. The scenarios of Figure 2 may be repeated from the point of view of *L2* and by a similar argument, it may be shown that *L2* must also obey the order received directly from the commander. But in Figure 2(b), the order is to *retreat*. Therefore in the scenario of Figure 2(b), *L1* will *attack* while *L2* will *retreat*, violating ***IC1***.

The scenarios of Figure 2 may be modified by substitution to prove the impossibility of satisfying ***IC1*** using *oral* messages with *3m* generals of which *m* are traitorous. The algorithm of Figure 3 will then require more stages (see [2]).

Now consider the case where $n = 3m + 1$, shown for $m = 1$ in Figure 4.

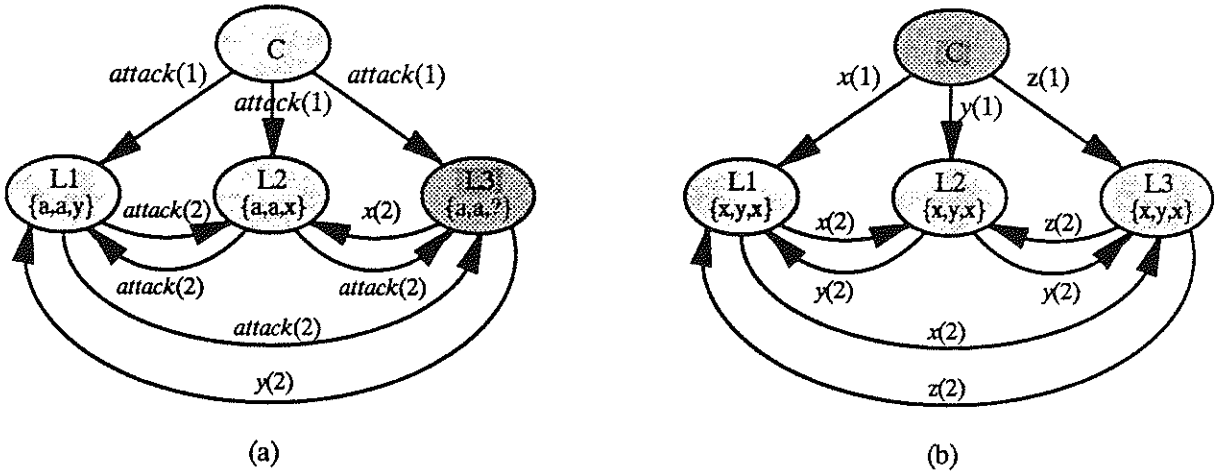

Figure 4 Sufficient Generals for Oral Messages

Each general executes the appropriate 2-stage algorithm of Figure 5.

In the scenario of Figure 4(a), the commander is loyal and sends the order to *attack* to all the lieutenants in stage 1. In stage 2, the loyal lieutenants *L1* and *L2* relay the order *he said attack* to each other and to the traitorous lieutenant *L3*, while *L3* sends an arbitrary order to *L1* (y) and *L2* (x). After stage 2, each lieutenant has a set of three opinions. Each lieutenant may then perform a majority vote on the set to reach a

decision. Both the loyal lieutenants have a majority of *attack* opinions and will therefore reach the same decision, to *attack*, thereby satisfying *IC1* and *IC2*.

```
process commander
    var v:order
    cycle
        formulate_order(v)
        send_to_all_lieutenants(v)
    end
end

process lieutenant
    var v, v', v'', v_c:order
    cycle
        within t do
            receive_from_commander(v)
        timeout
            v = retreat
        od
        send_to_other_lieutenants(v)
        par
            within t do
                receive_from_other_lieutenant(v')
            timeout
                v' = retreat
            od
            within t do
                receive_from_other_lieutenant(v'')
            timeout
                v'' = retreat
            od
        endpar
        v_c = maj( v, v', v'')
    end
end
```

Stage 1

Stage 1

Stage 2

Figure 5 Oral Message Algorithms

Now consider the scenario of Figure 4(b), where the commander is traitorous and sends conflicting orders $\{x,y,z\}$ to its lieutenants in stage 1. In stage 2, the loyal lieutenants each correctly relay the order they received in stage 1. After stage 2, each lieutenant has a set of three opinions. Each lieutenant may then perform a majority

vote on the set to reach a decision. All the lieutenants have the same set of opinions $V = \{x,y,z\}$ and will therefore reach the same decision (**maj** $\{x,y,z\}$) thereby satisfying *IC1*. Condition *IC2* does not apply because the commander is traitorous.

The scenarios of Figure 4 may be modified by substitution to prove the validity of satisfying *IC1* and *IC2* using *oral* messages with $3m + 1$ generals of which m are traitorous. The algorithm of Figure 5 will then require more stages (see [2]).

2.4.4 *A Solution using Signed Messages*

As was shown by the scenarios of Figure 2 and Figure 4, it is the ability of the traitorous general to lie which makes the *Byzantine Generals Problem* impossible for $n \leq 3m$. This ability may be restricted by enabling the generals to *sign* a message under the assumption that:

A4 (a) A loyal generals *signature* cannot be forged.

(b) Any alteration of the contents of a *signed message* can be detected.

(c) Any general can verify the *authenticity* of another generals *signature*.

A solution to the *Byzantine Generals Problem* now exists for the case $n \geq m + 2$.

Consider the case where $n = 3$ and $m = 1$ as shown in Figure 6.

Each general executes the appropriate 2–stage algorithm of Figure 7.

In the scenario of Figure 6(a), the commander is loyal and in stage 1 orders *attack*. In stage 2, the traitorous lieutenant *L2* tries to lie to the loyal lieutenant *L1* by saying that he received the order to *retreat*, as he did in Figure 2(a). However, by assumption *A4*, *L1* will now be able to detect a corruption of the message which may then be discarded. After stage 2, the loyal general *L1* will hold the single command *attack*:C sent directly by the loyal general. *L1* may then obey the command, satisfying *IC1* and *IC2*.

In the scenario of Figure 6(c), the commander is loyal and in stage 1 orders *attack*. In stage 2, the traitorous lieutenant *L2* refuses to relay an order to the loyal lieutenant *L1*.

After stage 2, the loyal general *L1* will hold the single command *attack*:C (as in Figure 6(a)) sent directly by the loyal general. *L1* may then obey the command, satisfying *IC1* and *IC2*.

In the scenario of Figure 6(b), the commander is traitorous and orders *L1* to *attack* while ordering *L2* to *retreat* in stage 1. Both orders carry the valid signature of the commander. In stage 2, the loyal lieutenants correctly relay the order they received in stage 1. After stage 2, each loyal lieutenant has two conflicting commands bearing the valid signature of the commander. They may therefore deduce that the commander is faulty, and assume the default order to *retreat*, thereby satisfying *IC1*. Condition *IC2* does not apply because the commander is traitorous.

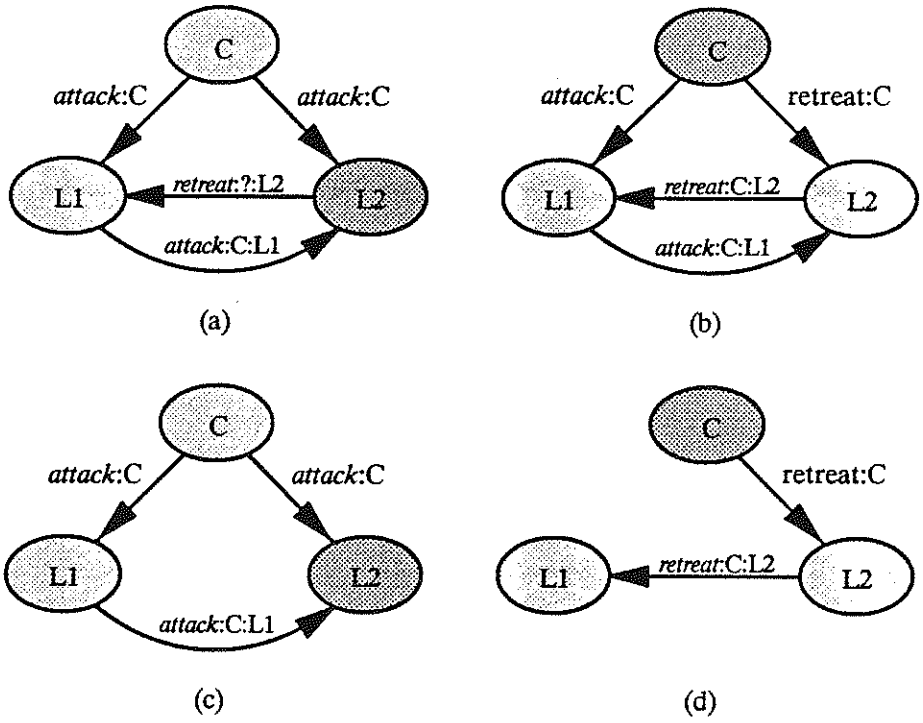

Figure 6 Sufficient Generals for Signed Messages

In the scenario of Figure 6(d), the commander is again traitorous and orders *L2* to retreat in stage 1, while refusing to send an order to *L1*. In stage 2, *L1* relays the command to *L2* but *L2* has nothing to relay to *L1*. After stage 2, both the loyal

lieutenants will hold the single command retreat:C sent by the traitorous general. Both

```
process commander
    var v:order
    cycle
        formulate_order(v)
        append_signature_to(v)
        send_to_all_lieutenants(v)
    end
end

process lieutenant
    var v, v', v_c:order
    cycle
        within t do
            receive_from_commander(v)
        timeout
        od
        if received_from_commander(v) then
            if not authentic(v) then discard(v)
            else
                append_signature_to(v)
                send_to_other_lieutenant(v)
            endif
        endif
        within t do
            receive_from_other_lieutenant(v')
        timeout
        od
        if received_from_other_lieutenant(v') then
            if not authentic(v') then discard(v)
            endif
        endif
        if received_from_commander(v)
            or received_from_other_lieutenant(v')
        then v_c = decide_order(v,v')
        else v_c = retreat
        endif
    end
end
```

Stage 1

Stage 1

Stage 2

Figure 7 Signed Message Algorithms

may then obey the order, satisfying *IC1*. *IC2* does not apply because the commander is traitorous.

Finally, if the commander is traitorous and refuses to send any order (not shown in Figure 6), both the loyal generals will obey the default order, to *retreat*.

The scenarios of Figure 6 may be modified by substitution to prove the validity of satisfying ***IC1*** and ***IC2*** using signed messages with $m + 2$ generals of which m are traitorous. The algorithm of Figure 7 will then require more stages (see [2]).

2.4.5 *Summary*

It has thus been shown that *interactive consistency*, satisfying the conditions ***IC1*** and ***IC2***, may be maintained in the presence of m traitors when there are are at least $3m + 1$ generals (oral messages) or $m + 2$ generals (signed messages). Therefore the conditions ***cond1'*** (and hence ***cond1***) and ***cond2*** of the original problem are satisfied under the stated fault assumptions. Since every loyal general receives the same set of opinions (as a lieutenant to each general in turn), by applying **maj** *V(i)*, they will all reach the same decision (satisfying ***condA***), either to *attack* or *retreat*. If there are at least $2m + 1$ generals, then the loyal generals will be in a majority and a small number of traitorous generals (any minority) cannot cause this to be a bad decision (satisfying ***condB***). Thus to satisfy ***condA*** and ***condB***, there must be at least $3m + 1$ generals using oral messages or $2m + 1$ generals ($2m + 1 \geq m + 2$) using signed messages.

2.5 Total Ordering

The discussion that follows shows how ***O1*** and ***O2*** may be met by the total ordering of client requests.

A client must be able to assign a unique identifier to each request it produces. This identifier will consist of the identity of the client c, the identity of the state machine sm along with a timestamp t which is unique for the given pair (c, sm). Requests from a given client c may be then ordered according to timestamp at sm, satisfying ***O1***.

However, if requests from multiple clients are to be ordered to satisfy *O2*, the timestamp attached to a given request *r* by a client *c* must be greater than that carried by any communication previously received by *c* (a communication could be received by *c* if *c* was itself to act as a state machine to other clients).

If two requests are received by *sm* bearing identical timestamps then their order of acceptance is unimportant and may be decided by some deterministic algorithm to ensure the replicas make a consistent choice. The choice may be based on the unique client identifier (the clients thus assume a priority).

In the following sections, two total ordering methods will be presented. The first, which uses logical clocks, does not require processor clock synchronization but has difficulty in determining when a request may be accepted (stability) in the presence of faulty processors. The second, which uses approximately synchronized real-time clocks, does not suffer from the stability problem associated with logical clocks but incurs the unavoidable overheads associated with processor clock synchronization.

2.5.1 *Logical clocks*

A logical clock [5] is a mapping F of events E to integers I:

$$I_n = F(E_n)$$

In the state machine context, an event may be the issue of a request or the receipt of some communication. For any two distinct events E_j, E_k:

$$F(E_j) \neq F(E_k)$$

Further, if E_j could have caused E_k then:

$$F(E_j) < F(E_k)$$

Logical clocks may then be implemented using counters. Every client c_m has its own counter C_m whose values are taken from the set of integers. Every request issued by a client carries the current value (timestamp) of its counter and the identity of the client. The counter is modified according to to counter-update rules:

CU1 C_m is incremented after after each event at c_m.

CU2 Upon receipt of a message with timestamp τ, C_m is updated further:

$$C_m = \max(C_m, \tau) + 1$$

The logical clock (timestamp + client identifier) may then be used to impose a total ordering on events, and hence requests to a given state machine *sm* may be ordered, satisfying ***O1*** and ***O2***.

However, there is a problem. To satisfy ***O1*** and ***O2***, a state machine cannot accept a given request r_s until it is no longer possible for another request to arrive bearing a lower or equal timestamp. This is termed stability and occurs only under the following conditions:

CON1 The state machine has received at least one request from every client in the system.

CON2 The timestamps carried by the latest request from each other client are all greater or equal to that carried by r_s.

If any client omits to send a request for some period, r_s cannot become stable, so all clients must periodically make null requests to all state machines. However, the failure of a single client could prevent the issue of null request and hence prevent r_s becoming stable.

This problem can be solved in the presence of fail-stop failures by *sm* setting a separate timeout on the receipt of the null request from each client. A timeout would

indicate failure of a client. No further requests could be received from that client so r_s could become stable.

This problem can only be solved in the presence of arbitrary (Byzantine) failures if the non–faulty state machines can agree on whether or not the timeout has occurred then ignore all further requests from the failed client.

2.5.2 *Approximately Synchronized Real–Time Clocks*

Each processor has its own real–time clock and the clocks of all processors are synchronized to within some known bound ϵ. Each clock increases monotonically in steps of δ_t. Every request issued by a client carries the current value (timestamp = τ) of its clock and the identity of the client.

O1 will be satisfied if a client makes no more than one request between successive clock ticks. *O2* will be satisfied if the clock synchronization error ϵ is less than the minimum time necessary to deliver a request (Δ_{min}) from client c to state machine *sm*. If Δ_{min} were less than ϵ, and the clock of *sm* lagged behind the clock of c by ϵ, then a request r issued at local clock time t_c ($= \tau$) could be delivered to *sm* at a its local clock time t_{sm} where:

$$t_{sm} < t_c$$

The state machine could then issue a request r' as a consequence of receiving r which carried a timestamp lower than that of r, violating *O2*.

As with logical clocks, to satisfy *O1* and *O2*, a state machine cannot accept a given request r_s until it is no longer possible for another request to arrive bearing a lower or equal timestamp (τ). If through clock synchronization the maximum time necessary to deliver a request (Δ_{max}) from client c to all non–faulty state machines is bounded, then r_s may be accepted when the local clock t_{sm} satisfies:

Stability 1 $\quad T_{sm} > \tau + \Delta_{max} + \epsilon$

This requirement forces the state machine to delay all requests by up to $\Delta_{max} + \epsilon$. The delay may be reduced if an additional stability test is included based on the conditions *CON1* and *CON2* specified for a system employing logical clocks.

Stability 2 A request r is additionally stable at state machine replica sm if a request bearing an equal or larger timestamp has been received from every other client in the system.

The first stability test can be made to tolerate a range of faults from fail–silent to arbitrary (Byzantine) if requests are delivered using an Atomic Broadcast Protocol as described in Section 5.3. In general, however, as the range of tolerable faults is increased, the time elapsed before a request becomes stable is also increased. The second stability test breaks down in the presence of faults as it did for logical clocks, described in the previous section. But since this is an optimization on the first test, no additional steps need be taken to make it fault–tolerant. In the presence of faults all requests would still become stable according to the first test.

2.6 Introducing Non-determinism into the State Machine Model

When replicating a state machine for fault-tolerance it is necessary to ensure that non-faulty replicas produce the same outputs. As has been shown, this is achieved by ensuring that:

P1 The outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.

Order Requests are processed in the same order by every non-faulty state machine replica.

P1 is achieved by processing requests sequentially and ensuring that each command is implemented by a deterministic program. Order is achieved by total message ordering, thereby depriving the state machine of any control over the selection of requests.

Most distributed systems intended for real-time applications employ mechanisms which are prohibited by the state machine model. For example, *P1* would be violated by internal concurrency, timeouts or asynchronous external events (interrupts), while Order could be violated by prioritized requests or timed access restrictions. If such distributed systems are to be replicated for fault-tolerance, the State Machine Model must be enhanced to incorporate these mechanisms.

To illustrate how such mechanisms could defeat successful voting, timeouts and prioritized requests will now be considered.

2.6.1 *Timeouts*

Suppose that a state machine has been designed to manage some external device, such as a robot arm. The state machine is replicated and the outputs of the replicas combined through voting by the arm control circuit. In response to a

move_to(new_position, *t*) request from a client, the state machine must perform some iterative calculation to decide the most efficient route (avoiding obstacles) then issue a sequence of outputs to move the arm to the new position before time *t*. The task of the state machine is thus to calculate the best route in the available time. The command required to service the request could be programmed as in Figure 8.

```
process state_machine
    var m:message
    cycle
        receiveany(m)
        switch (m.request)
        ...
        case move_to:
            calculate(route, m.new_position)
            deadline = m.t - time_needed_for_move
            while time < deadline
            do improve(route, m.new_position)
            output(route)
        ...
    end
```

Figure 8 The Non–deterministic State Machine

The total number of iterations performed by the state machine will depend on other processor activity and may vary between replicas. It is thus possible for the replicas to produce different opinions on the best route (output). This could defeat the voting mechanism, even in the absence of faults.

2.6.2 *Prioritized requests*

Suppose that the robot arm of Section 2.6.1 continuously receives independent move_to requests from a number of clients. At any point in time, there will be a number of requests pending. But their order in the queue may not correspond to the order of their deadlines, so instead of processing requests in the order dictated by the receiveany(m) function (total ordering), requests must be selected by the state machine, after taking into account the deadline associated with the move. Selecting

requests in this way effectively assigns a level of priority to each request according to its deadline parameter and requires a prioritized search of the queue of pending requests.

However, although requests are delivered to state machine replicas in the same order, the time at which a given request is delivered to a given replica relative to the time of its search may vary. So at the time of search, the replica queues may not be consistent. It is therefore possible that the highest priority request selected by one replica is not present in the queue of another replica at the time of its search. In this case, the other replica will select a different request so their outputs will differ, thereby defeating the voting mechanism, even in the absence of faults.

2.7 Summary

A fault-tolerant state machine can be implemented by replicating it and executing each replica on a distinct processor in a distributed system. For voting to be successful, non-faulty replicas must produce the same outputs. This may be achieved by the total ordering of requests to make request selection deterministic and by the prohibition of internal non-determinism.

However, most distributed systems employ mechanisms which would, if incorporated directly into the state machine model, introduce possible sources of non-determinism. This non-determinism could lead to a divergence of state among the state machine replicas. They could thereafter produce inconsistent outputs which would defeat the voting mechanism.

If non-deterministic behaviour is to be introduced into the state machine model, steps must be taken to ensure that state divergence does not occur, so that the replicas continue to produce consistent outputs. A solution will be presented in Chapter 6.

Chapter 3 : Fault-Tolerant Architectures

3.1 Introduction

Many fault tolerant distributed systems have been implemented under the assumption that processors have *fail-silent* semantics, that is they fail *cleanly* by just stopping [17][26][31][32][46][47][78]. Such a restricted view is hard to justify in computer systems intended for mission and life critical applications where failure probabilities in the range 10^{-6} to 10^{-10} per hour are often specified [28][37]. It is then preferable to design and implement such systems under a totally unrestricted fault assumption namely, that a failed processor can behave in an arbitrary manner. In the literature this failure mode is often referred to as the *Byzantine* failure mode [2] (see Section 2.4).

Replicated processing with majority voting, *N-modular redundant* (NMR) processing [1], is a technique which enables the construction of systems which tolerate Byzantine processor failures. NMR techniques have already been applied successfully to distributed and multi-computer systems [6][21][28][37][47] including real-time control applications such as railway signalling [79]. When the degree of replication is three the technique is known as *Triple Modular Redundancy* (TMR).

The communication of information between processors required to perform majority voting is similar to that required to solve the *interactive consistency problem* (see Section 2.4). However, there is one fundamental difference which affects the complexity of the solution and the minimum number of processor n necessary to tolerate m failures [30].

In the *interactive consistency problem*, loyal generals can legitimately form a different opinion on whether to attack or retreat. To ensure that all loyal generals make the same decision it is necessary to ensure that all loyal generals possess the same set of

opinions (V_i in Section 2.4) before performing a majority vote. This may be achieved by repeatedly applying stages 1 and 2 (more for $m > 1$) of the *Byzantine Generals Problem* solution, once for each general. The minimum number of processors n necessary to tolerate m failures is $n \geq 3m + 1$ using oral messages or $n \geq 2m + 1$ using signed messages as discussed in Section 2.4.

With NMR processing on the other hand, it may be assumed that all loyal generals share the same opinion (non–faulty processors produce the same output message). So, if the loyal generals are in a majority, their common opinion will form a majority, irrespective of the opinions of the traitorous generals. A majority vote may thus be performed after repeatedly applying only stage 1 of the *Byzantine Generals Problem* solution, once for each general. The minimum number of processors n necessary to tolerate m failures is $n \geq 2m + 1$ using oral messages (signed messages become necessary only if processors are not directly connected).

Thus when the problem of state divergence is disallowed or prevented, the *interactive consistency problem* cannot occur and such an NMR processing system may be constructed using only $2m + 1$ processors.

Several existing fault–tolerant systems will now be examined, with a view to showing how each prevents state–divergence.

The systems may be divided into four groups.

In the first group (Tandem, Delta–4 OSA passive model), processors are assumed to have *fail–silent* semantics, allowing passive replication techniques to be used (since outputs from a functioning processor are by definition correct), so non–determinism within process replicas cannot cause state divergence.

In the second group (SIFT, Mars, Delta–4 OSA active model), process replicas are constrained to be deterministic and cannot therefore cause state divergence. However,

in SIFT and Mars, communication between concurrently executing processes is asynchronous, another possible source of non–determinism which must be tackled if state divergence is to be prevented.

In the third group (FTMP, FTP, Sequoia, Stratus), process replicas operate in *lock–step* execution (micro–frame synchronization [35]). and special hardware is used to synchronize asynchronous events (eg. interrupts and timeouts) to the processor instruction stream so process replicas are interrupted at the same point in their execution. Therefore non–determinism within process replicas cannot cause state divergence.

In the fourth group (Delta–4 XPA, MAFT), agreement protocols are executed on behalf of the process replicas, to resolve non–determinism and prevent state divergence.

3.2 The Tandem Non–Stop System

The Tandem Non–Stop system [14][15][17][18][24][26] is a commercial distributed computing system for on–line transaction processing. When the system was designed in the mid 1970s the view was taken that active replication of processes through hardware redundancy would not be cost effective for that application. So, Tandem assumes that processors have *fail–silent* semantics, then uses passive replication to tolerate failures. A simplified representation of the Tandem architecture is shown in Figure 9.

A Tandem system consists of 2 to 16 processing modules connected via dual busses (The Dynabus). Each processing module consists of a single processor with its private memory. Multiple Tandem systems may be connected by a replicated fibre–optic ring to increase the number of processing modules. All inter–processor communication is performed by passing messages on Dynabus.

Each process is duplicated to form primary and secondary replicas. The primary replica *P* executes the algorithm of the process and periodically checkpoints its state to the secondary replica *S*. The secondary replica, which resides on another processor, uses these checkpoints to update its own state. The secondary replica does not execute the algorithm of the process and hence does not consume processor time (except when receiving checkpoints).

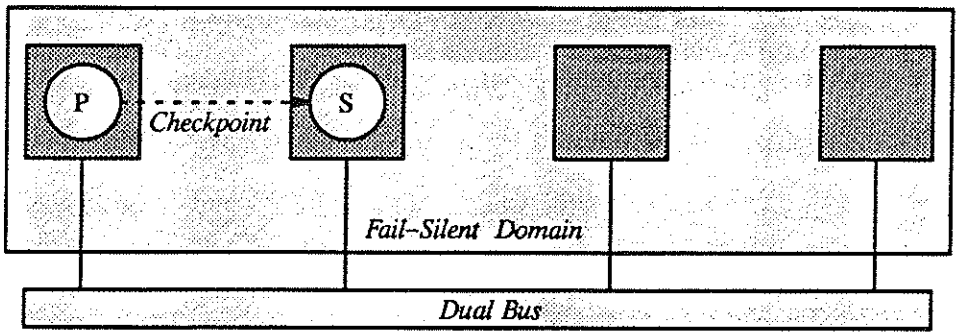

Figure 9 Tandem

Every processor periodically (once per second) transmits an *I'm alive* message which is monitored by all processors. It is assumed that the absence of such a message (detected by timeout) indicates the failure of that processor. When a failure is detected, processors participate in an interactive consistency algorithm to agree on the failure. Then each processor checks to see if it maintains a secondary replica for any primary on the failed processor. If it does, the secondary replica is activated, beginning execution at its last checkpoint. The secondary replica now becomes the new primary, but without a secondary of its own (failure of the new primary cannot be tolerated).

When the failed processor recovers or is replaced, it again periodically transmits an *I'm alive* message. This is detected by the new primary and causes a new secondary replica to be created on the formerly failed processor, thereby regaining fault–tolerance.

The secondary replica has its path of supposed execution imposed by the primary replica (through a sequence of checkpoints), so the path of execution of the primary may be non–deterministic without causing state divergence.

3.3 The Sequoia Computer System

Sequoia [24][25][27] is another commercial distributed computing system for on–line transaction processing which, like Tandem, uses passive replication to tolerate failures. However, whereas Tandem assumes that processors have *fail–silent* semantics, Sequoia assumes that processors have *fail–arbitrary* semantics and employs active replication (a processor pair) to implement a processor with real *fail–silent* semantics. A simplified representation of the Sequoia architecture is shown in Figure 10.

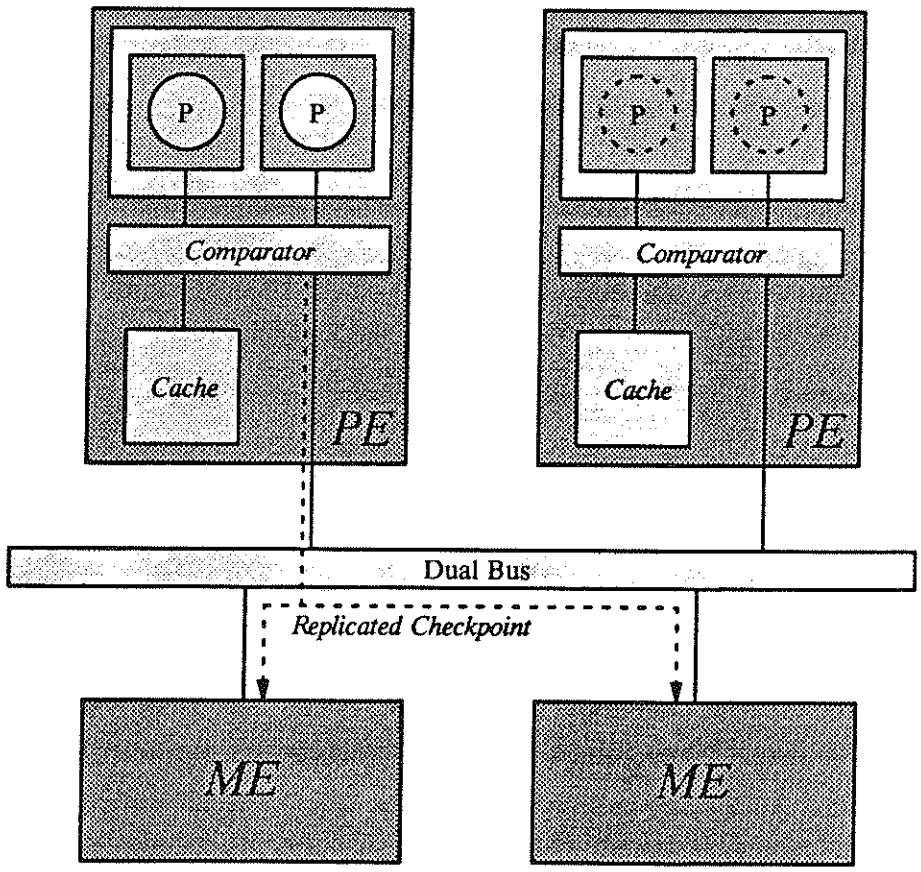

Figure 10 Sequoia

A Sequoia system consists of a number of processing elements (PEs) and memory elements (MEs), connected via a dual bus. Each PE is constructed using two processors which operate in *lock–step* execution. A comparator detects any difference in their behaviour and ensures that PEs have *fail–silent* semantics.

Each PE contains a cache memory. A process *P* executing on a PE uses this cache memory to store its data; all read and write requests operate on the cache. Periodically, the contents of the cache are flushed to two distinct MEs, thereby establishing a *replicated checkpoint*.

The code of a process is stored in an ME. If a PE fails, each process residing on the failed PE is restarted on another PE, beginning execution at its last checkpoint.

When implementing a PE with *fail–silent* semantics, only the failure of a whole processor is considered. Failures of MEs or busses, on the other hand, are dealt–with on a much finer granularity.

Data is protected by error–detecting codes while stored in a PE cache, in an ME or while in transit over a bus. The hardware implementing storage and data paths is partitioned such that no two bits of the same byte have a common component. Thus failure of a single component can only produce a single–bit error in any byte and all such errors are detectable. If an error is detected, the byte may be discarded and the component which sent it then has *fail–silent* semantics.

If an ME fails, a PE can always obtain data from the other ME to which it checkpoints. Read–only data is treated as a special case; it is not checkpointed. Consequently, it is stored on only one ME. If that ME fails, the data is restored from disc.

All Sequoia components thus have *fail–silent* semantics, either through active replication (PEs) or error detection (MEs and busses). Replication of all code and data, using checkpoints and disc back–ups, ensures that the execution of a failed process may be resumed elsewhere.

Processors within a PE operate in *lock–step* execution and receive exactly the same input data. External events are synchronized to this execution, so non–determinism within process replicas cannot cause state divergence among process replicas.

In the event of a failure, execution of a process is resumed by another PE. The new process inherits its history (its imaginary path of execution) from the old process via the latest checkpoint, so the path of execution of the process may be non–deterministic without causing state divergence.

3.4 The Stratus Computer System

The Stratus computer system [15][16][18][24] is another commercial distributed computing system for on–line transaction processing. Stratus assumes that processors have *fail–arbitrary* semantics and employs active replication (a processor pair) to implement a processor with real *fail–silent* semantics (like Sequoia). Active replication is then used again to mask the *fail–silent* failure of a processor pair.

A Stratus system consists of a number of processing modules (PM) connected by a dual bus called StrataLINK, as shown in Figure 11.

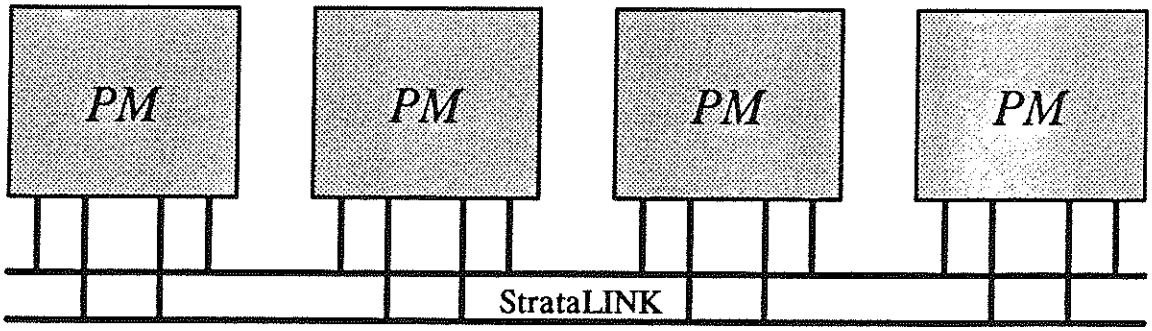

Figure 11 The Stratus Computer System

Each processing module consists of two identical processing elements (PE), connected by a dual bus called StrataBUS, as shown in Figure 12.

Each processing element consists of a number of components (CPU, memory etc) which constitute the smallest replaceable unit (SRU) of the system.

A CPU component is constructed from two identical processors operating in *lock–step* execution (like Sequoia). The outputs of the two processors are compared in hardware. As long as they match, outputs are gated onto StrataBUS. If a disagreement is detected, the outputs are not propagated, the component is isolated from StrataBUS and thus has *fail–silent* semantics.

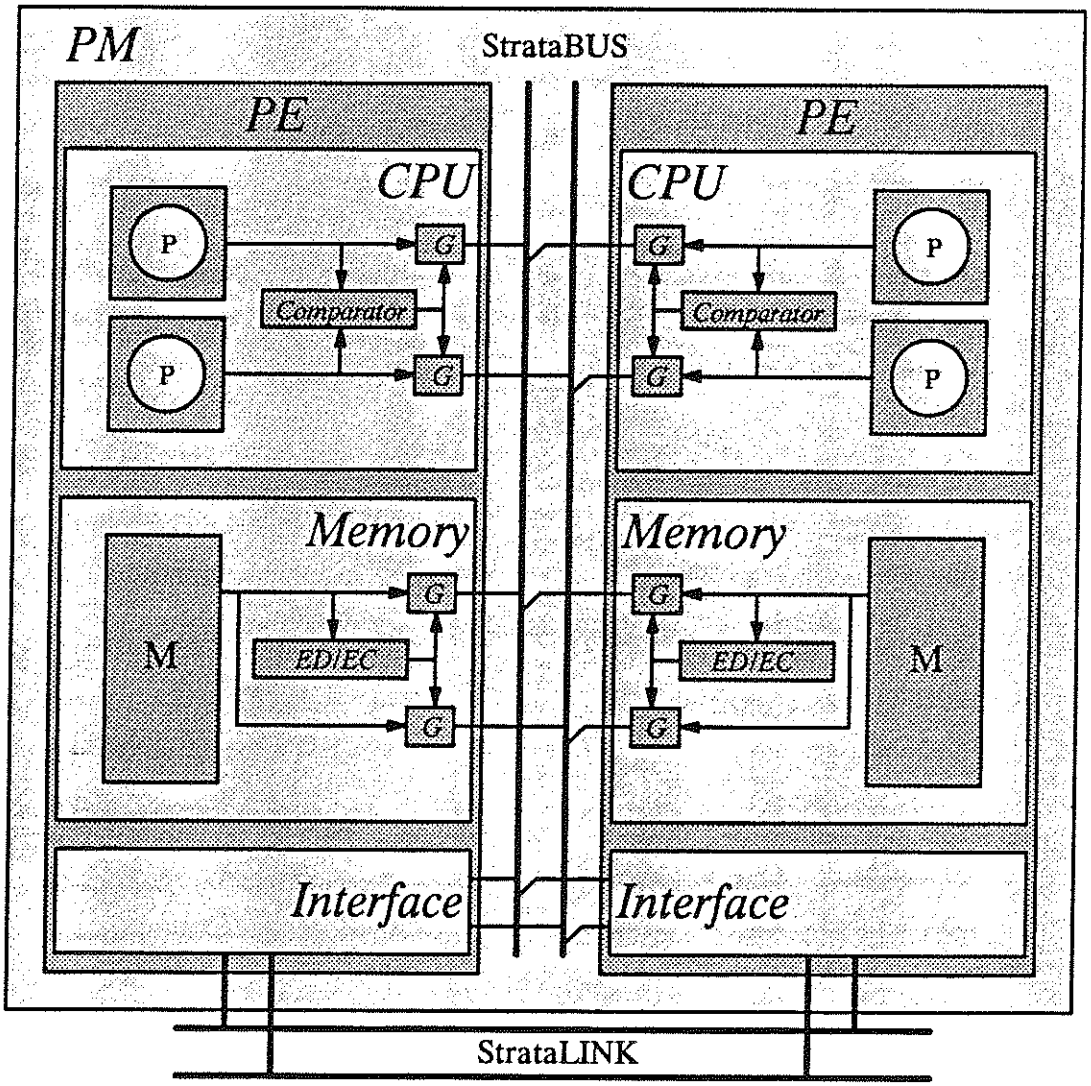

Figure 12 A Stratus Processing Module

A memory component contains error detection/correction circuits which prevents the propagation of erroneous data. A memory component therefore also has *fail–silent* semantics.

Stratus may also contain a number of other component types (not shown) such as disc controllers. In all cases, components have *fail–silent* semantics.

When a component in a *PE* issues/receives a message on StrataBUS, the corresponding component (in the other *PE* of the same *PM*) also issues/receives the same message (in the absence of failures). So when a single failure occurs, there is always at least one message issued/received and the failure is masked.

The processors which make–up the two CPU components in a processing module are fed exactly the same inputs in the same order in *lock–step*. Thus all four processors in a processing module operate in *lock–step* execution. Processes may therefore contain non–determinism without causing state–divergence.

3.5 SIFT

SIFT (*Software Implemented Fault Tolerance*) [17][19][28] is an ultra–reliable computer for critical aircraft control applications.

The physical architecture of the SIFT system is shown in Figure 13. It is composed of a number of processors P_i, each with its own memory M_i. Processors and memory are connected to a number of data busses. A processor may read or write its own memory but may only read the memory of other processors.

Every processor executes one or more processes. Every process is replicated and each replica executed on a different processor. Processes are iterative in nature. Each process repeatedly inputs data, performs some calculation and outputs the results. Data produced by a process replica is deposited in its own memory. Data to be input by

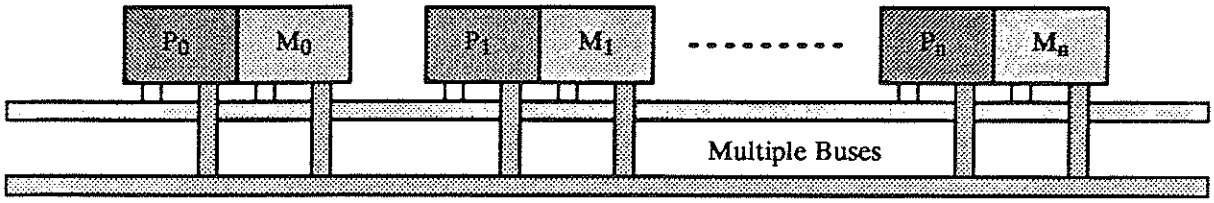

Figure 13 The Physical Architecture of SIFT

a process is read from the memory of several replicas. Each read operation is performed over a different bus, and the results combined through majority voting to produce a single input to the process. In this way, the failure of a single process replica (through the failure of its host processor) or the failure of a bus will be masked.

An iteration rate is specified for each process. The scheduler guarantees that a particular iteration of a task will be completed within the *time frame* for that iteration. Execution may take place at any point during the *time frame* but the results will be available by the end. Now suppose that a process P_a provides input data for another process P_b and that both processes have the same iteration rate, Figure 14 depicts four successive *time frames* in their execution. In some *time frames* P_a will complete before P_b and in others the converse will be true and the order may vary among the replicas of a process. If P_b were to take its data directly from P_a, this data could be from the same iteration (*time frames* 1 and 2) or from the previous iteration (time frames 3 and 4), as shown by the curved arrows in Figure 14. Since the order of execution may vary among replicas, some replicas will use data from the same iteration and some from the previous iteration. Unchecked, this could lead to state divergence within the replicas.

The problem is solved by the double-buffering of results passed from P_a to P_b, as shown in Figure 15. During odd-numbered *time frames* P_a outputs data to *buffer 1* while P_b inputs data from *buffer 2*. During even-numbered *time frames* P_a outputs data to *buffer 2* while P_b inputs data from *buffer 1*. In this way, the replicas of P_b will all use

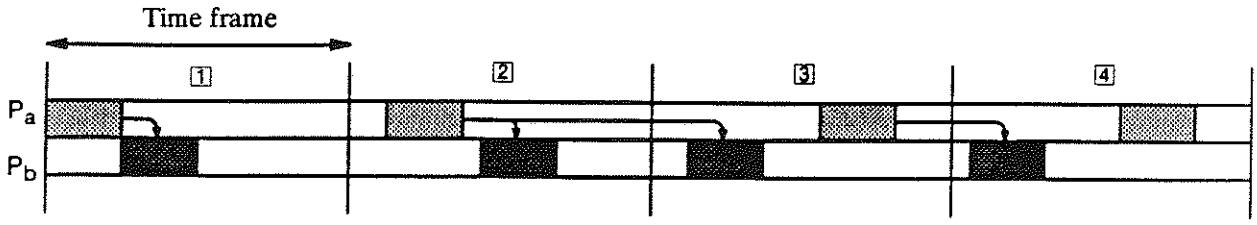

Figure 14 Process Execution

the data produced by P_a in the previous iteration and state divergence will be prevented.

The clocks of all processors are *loosely* synchronized to within 50μs (clock synchronization is another form of the *interactive consistency problem*, discovered and solved by the SIFT designers). So the time frames of different processors will also be synchronized to within 50μs. Now suppose the clock of P_b is running 50μs faster than that of P_a and P_a outputs its data to the buffer right at the end of a *time frame* (as determined by the clock of P_a). If P_b inputs its data from the buffer right at the start of a *time frame* (as determined by the clock of P_b), it could precede the data output by 50μs and the data would thus be two iterations old. It is necessary therefore to separate the input and output of data by, for example, preventing the inputting of data until at least 50 μs into the *time frame*. The proportion of the *time frame* available for input is then reduced by 50 μs.

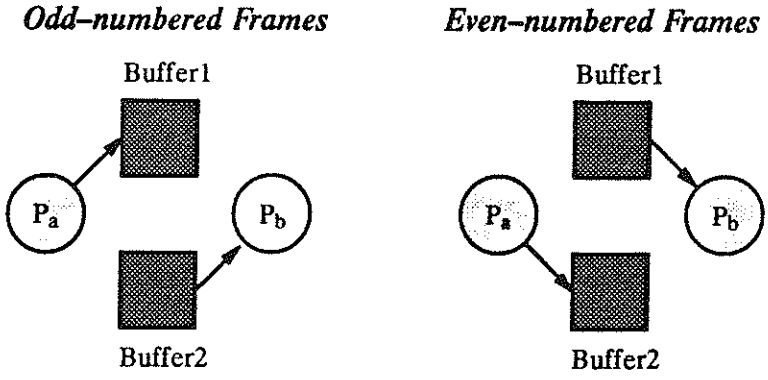

Figure 15 Double–buffering

SIFT processes must be deterministic. However, non-determinism may be incorporated into process scheduling by synchronizing asynchronous events to *time-frame* boundaries. In this way, either all process replicas will see a given event at the start of their execution, or none will.

With the bus-based SIFT architecture, data values are fetched from other processors and voted when required, incurring a delay. In a development of SIFT [29], the data is broadcast and voted in advance. So when required, it is available locally without a delay.

The double-buffering scheme of Figure 15 is not sufficient and must be expanded, as shown in Figure 16.

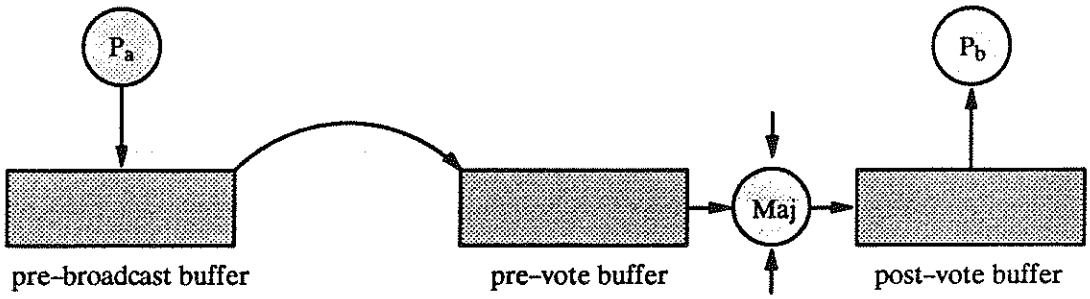

Figure 16 Modified Buffering

Now suppose that a process P_a provides input data for another process P_b. P_a places its data in the *pre-broadcast buffer* some time during its *time-frame* of execution. At the end of that *time-frame*, the data is broadcast to the processor executing P_b, where it is placed in a *pre-vote buffer*. When a majority of data copies have been received from replicas of P_a, the data is placed in the *post-vote buffer*. The broadcasts and voting take place during the *time-frame* following that in which P_a is executed, so the data is not available to P_b until the end of that *time-frame*. P_a and P_b should not therefore be scheduled for adjacent *time-frames* unless additional double-buffering is available.

3.5.1 *SIFT – An Extension*

A number of modifications to the SIFT strategy were proposed in [80] in order that some of its disadvantages may be overcome while still preventing state divergence. In both the modifications presented in this section, the process need not be iterative and no planned scheduling is necessary. But the rather restrictive assumption is made that a message cannot be lost, only corrupted.

In the first modification, two additional assumptions are made:

A1 A process P_a may send a message to the replicas of another process P_b in an indivisible way.

A2 The replicas of P_b process messages in an identical order.

The assumptions can be satisfied by centralizing communication. The multiple bus structure of SIFT is replaced by a single bus. To satisfy *A1*, a message is broadcast to either all of the replicas of P_b or none of them (bus failures are not considered). To satisfy *A2*, a replica of P_b will attach a *timestamp* to every message it receives and process messages from various sources in order of their *timestamps* (oldest first). Since the *timestamps* are only used by the processor which appended them, their absolute value is unimportant and hence there is no need for the synchronization of processor clocks.

It may be noted that the author has in fact described an *atomic broadcast* (see Section 5.3) which relies on trusted hardware for its implementation (the bus). The assumptions *A1* and *A2* are satisfied by the ***atomicity*** and ***order*** properties of the *atomic broadcast*. The third property of the *atomic broadcast*, ***termination***, is not considered. The replicas of P_a will each try to broadcast the same data, so in the absence of faults, each replica of P_b will receive three copies of every message and a mechanism for duplicate removal is necessary.

Although this solution removes some of the restrictions of the original SIFT approach to preventing state divergence, the requirement for a single bus creates a single point of failure and does not lend itself to the construction of massively parallel systems.

In the second solution, the need for a single bus is removed. A process P_b no longer has direct access to the output data of P_a. Instead, the replicas of P_b periodically perform an *interactive consistency* algorithm to agree on a set of messages which have arrived at or are about to arrive at P_b. They then independently order the set using some

deterministic algorithm, appending a sequence number to each message and processing them in order of sequence number.

Any implementation of the *interactive consistency* algorithm would have to solve the *Byzantine Generals Problem* of Section 2.4.2. In fact, the second solution to preventing state divergence is similar to ours except that the messages are *atomically broadcast* periodically in *batches* (instead of singly in advance), sequence numbers are used to order messages (instead of a *queue*), the process model is limited to *blocking* input (see Section 6.2.1) and the clocks of all processors must be synchronized.

3.6 Mars

The Maintainable Real-Time System (Mars [31][32][33]) was designed for real-time control applications where reliability, availability and safety are of paramount importance. The simplified physical architecture of a Mars cluster is shown in Figure 17.

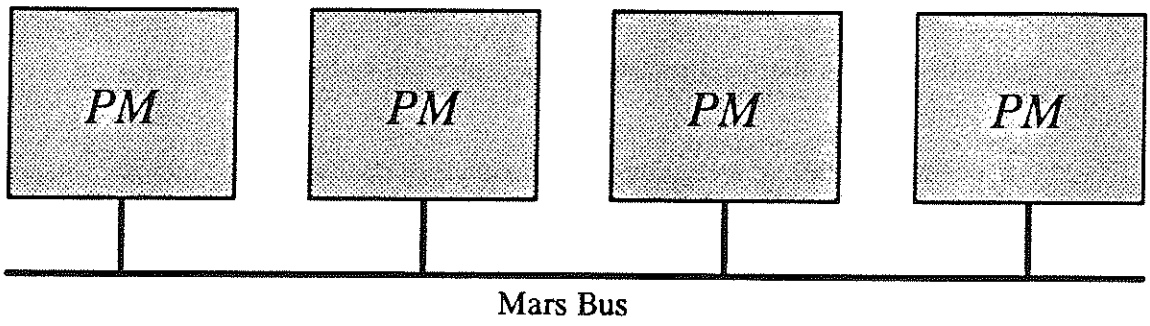

Figure 17 A Mars Cluster

It consists of a number of processors which are assumed to have *fail-silent* semantics, connected via a TDMA bus, called Mars Bus. A number of Mars clusters may be connected in a hierarchical structure (not shown).

The clocks of all processors are synchronized to create a fault-tolerant global timebase called *system time*. The scheduling of tasks is pre-planned and conforms to

one of a finite set of schedules, which may be changed to give the system a new *phase* or *mode*.

Processes communicate using *state messages* and *event messages*. The semantics of a state message is similar to that of a global variable. That is, a new version of a particular state message updates (overwrites) an old one and state messages are not consumed when read. The semantics of an event message is similar to that of a *conventional* message. That is, event messages are queued and an event message is consumed when read.

All messages carry a validity timestamp which, when it expires, causes a state message to become invalid or an event message to be discarded.

Mars uses active replicateion. If processes were allowed immediate access to messages, state divergence could occur, because message arrival is asynchronous with respect to process execution. Instead, all messages are buffered. Periodically, a clock–driven interrupt routine is executed which acts on all buffered messages to create a new *image set*, representing a new global state.

Process scheduling, bus scheduling and clock–driven interrupts are all synchronized. So when process replicas perform a *read* operation, they will all read from the same image set and state divergence will be prevented.

3.7 FTMP

The *Fault Tolerant Multiprocessor* (FTMP) [17][19][34] was designed as part of a commercial aircraft *fly–by–wire* program. The physical architecture of an FTMP system is shown in Figure 18.

Like SIFT, it is composed of a number of processors P_i, each with its own memory M_i, collectively termed a processing module. Unlike SIFT however, only processors are

connected to the multiple data busses, the memory remains private to its attached processor. Instead, there are a number of global memory modules, also connected to the multiple busses and accessible to all processors. Any module, whether a processing or memory module, may receive data on all busses but may send data on only one (the choice of this bus is quasi–static but reconfigurable). Modules are logically grouped into triads such that each member sends its data on a different bus.

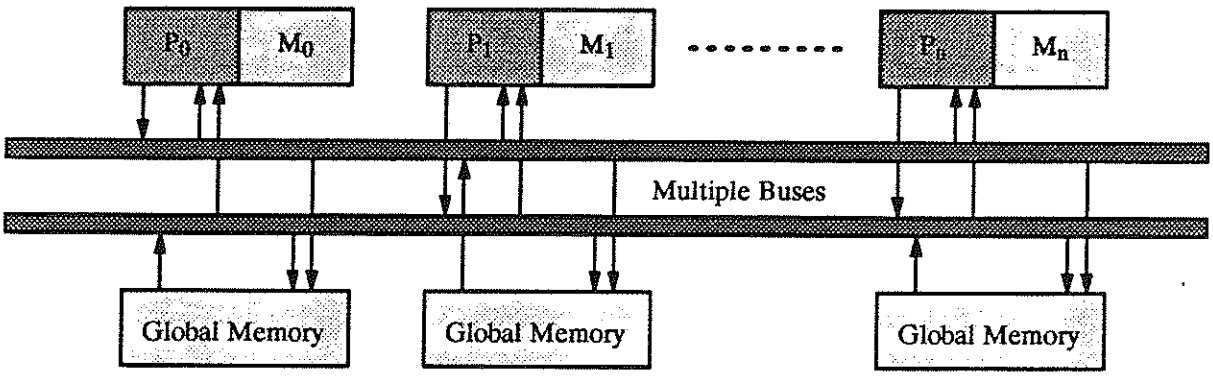

Figure 18 The Physical Architecture of FTMP

Every processor executes exactly one process (at any given point in time). Every process is replicated (to degree of three) and each replica executed on a different member of a processor triad. When a process triad sends a message to another triad, three messages are sent, one from each replica. These three messages will necessarily be sent on different busses (a bus triad). Each replica in the destination triad receives all three messages, one from each bus, combining them in hardware to form a majority. As with SIFT, the failure of a single process replica (through the failure of its host processor) or the failure of a bus will be masked. Although the majority is formed by hardware, each member of the destination triad has its own voter. So a failure of the voter will be treated as a failure of the attached module, ie there is no *trusted* hardware.

All module clocks are *tightly* synchronized using a fault–tolerant hardware clock. So the processes of a triad remain in *lock–step* execution and will perform data input and output requests synchronously. Although process triads operate independently, their

access to the multiple busses is strictly controlled such that the processors of a triad are granted access simultaneously and exclusively, even if the two bus triads concerned have no common members (as shown in Figure 19 for process triads P_a, and P_b). That is, at any point in time, there may only be one active bus triad.

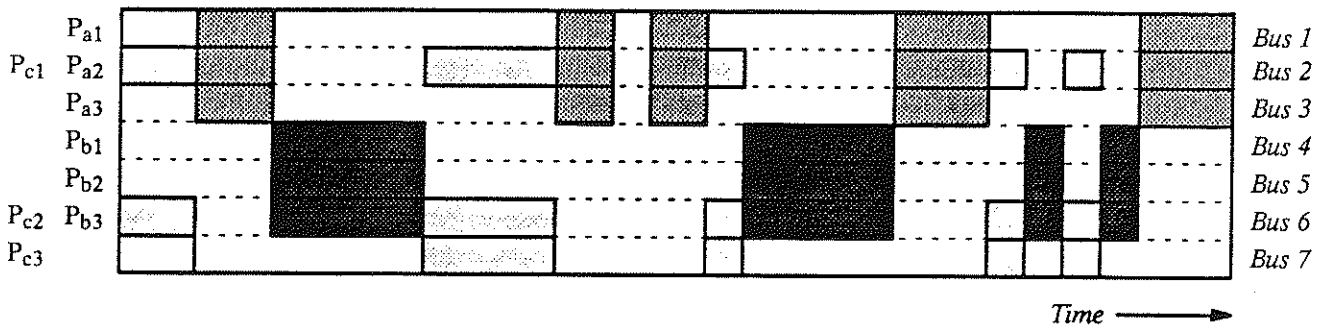

Figure 19 Bus Activity

Although there are multiple busses as in SIFT, the enforced sequentially of bus traffic limits the bandwidth to that of a single bus no matter how many busses are available. This bandwidth would be insufficient for large systems.

Bus transfers thus constitute an ordering mechanism which eliminates the possibility of state divergence within *deterministic* process replicas by centralizing communication in a way reminiscent of that described in Section 3.5.1. Note that the system behaves as though three *atomic broadcasts* were being performed in parallel, one from each of the replicas.

Because the members of a processor triad operate in *lock–step* execution, processes may contain *non–determinism* without causing state divergence.

3.8 FTP

The *Fault Tolerant Processor* (FTP) [19][35][36] is a derivative of FTMP designed for nuclear power plant applications. The system consists of four processors which are fully connected as shown in Figure 20.

This quadruplex configuration is used in preference to the triplex configuration of FTMP to improve the reliability. The replicas periodically output data. This data is voted in hardware to mask failures and the result passed back to the replicas as input. The processors are *tightly* synchronized by synchronizing their clocks, so process replicas operate in *lock–step* execution. Asynchronous *events* are synchronized to the

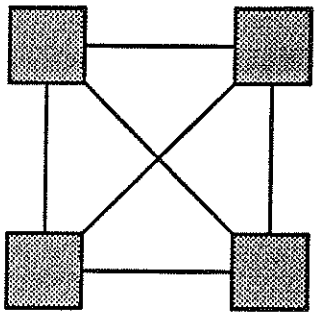

Figure 20 The Architecture of FTP

processor instruction stream, so process replicas *see* the *event* at the same point in their execution. Processes may thus contain non–determinism without causing state divergence.

3.9 MAFT

The Multicomputer Architecture for Fault–Tolerance (MAFT) system [39][40][41][42][43][44][45] is similar to SIFT in that it consists of a number of semi–autonomous processors connected via multiple busses and its processes are periodic in nature. However, MAFT allows some non–periodic behaviour through conditional branch scheduling and design diversity of both hardware and software through median voting algorithms.

The architecture of the MAFT system is shown in Figure 21.

Each MAFT processor consists of an Application Processor (AP) and a hardware intensive Operations Controller (OC). The APs which may be heterogeneous, execute

application processes. The OCs execute all the voting and consistency algorithms and are responsible for process scheduling on the APs.

Processes are replicated to a degree specified by the application and each replica executed on a distinct AP (as an alternative, the replicas may be diverse).

OCs communicate using messages. Each OC transmits on a different bus but may receive on all buses. Every transmission by an OC thus constitutes a broadcast (non–atomic) to all other OCs.

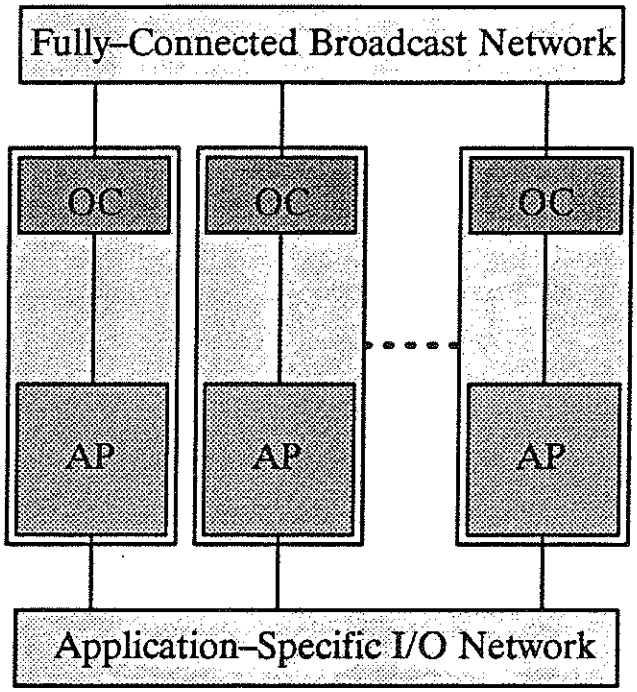

Figure 21 The MAFT Architecture

An application process consists of an indivisible block of code which must be executed without interruption on a single AP. An iteration rate is specified for each process. The inverse of the maximum iteration rate is called the atomic period. The inverse of all iteration rates must be binary multiples of the atomic period. Several different short processes may be executed strictly sequentially during the same atomic period, where time permits.

The atomic period is further divided into an integral number of sub–atomic periods. The execution of a process always starts on a sub–atomic boundary.

The clocks of the OCs are loosely synchronised [45] through the exchange of system state (SS) sync messages (and presync messages) whose transmission marks an atomic boundary. Thus the atomic and sub–atomic boundaries of all OCs are also loosely synchronised.

When an AP completes the execution of a process, it sends a branch condition (BC) to its OC. The OC then broadcasts a process completed/started (CS) message containing BC to all other OCs.

On each sub–atomic boundary, an OC broadcasts an interactive consistency (TIC) message consisting of two bytes. The TIC reflects the scheduling activity (as seen by that OC) of all processors during the previous sub–atomic period. The first byte (TC) indicates those processors from which a CS message was received. The second byte (BC) indicates the branch conditions contained within those CS messages.

The CS and TIC messages form the basis of an interactive consistency algorithm which ensures that all nodes reach agreement on the existence and content of each CS message transmitted, even in the presence of a single malicious (Byzantine) failure. Each OC may then execute any deterministic scheduling algorithm which uses the TIC information and consistent scheduling will be maintained.

The TIC allows the scheduler to determine whether or not a particular process has completed its execution. The application may therefore specify simple dependencies (process B to be scheduled only when process A has been completed). More complex dependencies may be specified using an AND–FORK (processes B and C to be scheduled only when process A has completed) or an AND–JOIN (process D to be scheduled only when processes B and C have completed).

The TIC also allows the scheduler to use consistent BCs generated by the APs to implement conditional branching using an OR–FORK (If process A returns true then schedule process B else schedule process C) or an OR–JOIN (Schedule process D when either process B or process C has completed).

The OCs maintain a form of distributed shared data memory on behalf of their AP processes so a request for data from an AP may be satisfied immediately.

When an AP process wishes to modify part of this data, it sends the new value to its OC. The OC forms a data message by appending to this value a data identification descriptor (DID) which uniquely labels the data in the message then broadcasts the message to all other OCs.

When An OC receives a data message, it performs an acceptance test based on a range of acceptable values particular to that DID. If the data message fails the test, it is discarded and an error signal raised. If it passes the test, it is forwarded to a voter.

The voter performs an instant vote using the new data message and any other data messages previously received from other replicas. The voting algorithm is based on median voting to cope with the inconsistent values produced by diverse software implementations of the process replicas. It is conceivable that a faulty OC could broadcast inconsistent values to other OCs. This could result in a difference in the outputs of their voters. But since all data messages must undergo an acceptance test, voter outputs would be bound to an acceptable range of values. Thus approximate agreement is achieved in the presence of arbitrary (Byzantine) failures.

The MAFT OC uses an interactive consistency algorithm to determine when a given process replica has terminated, and to agree on the BC it returned. This information is then used to perform non–periodic scheduling. The value of BC returned by a given process replica need not be consistent among non–faulty replicas. So although the execution of a given process replica must be deterministic, the calculation of its BC

need not and hence the next process to be scheduled may *depend on time and any other activity in the system* (see Section 2.2). In this way, the sequence of processes executed by MAFT may be non–deterministic without causing state divergence among process replicas.

3.10 Delta–4 OSA

The aim of the Delta–4 project [46][47] is to define an open, fault tolerant, distributed systems architecture (OSA). Processors may be heterogeneous and may have *fail–silent* or *fail–arbitrary* semantics. Every processor is connected to a single dependable *local area network* (LAN) via a *network attachment controller* (NAC) which has *fail–silent* semantics. Figure 22 shows an example system of three *fail–silent* processors and three *fail–arbitrary* processors.

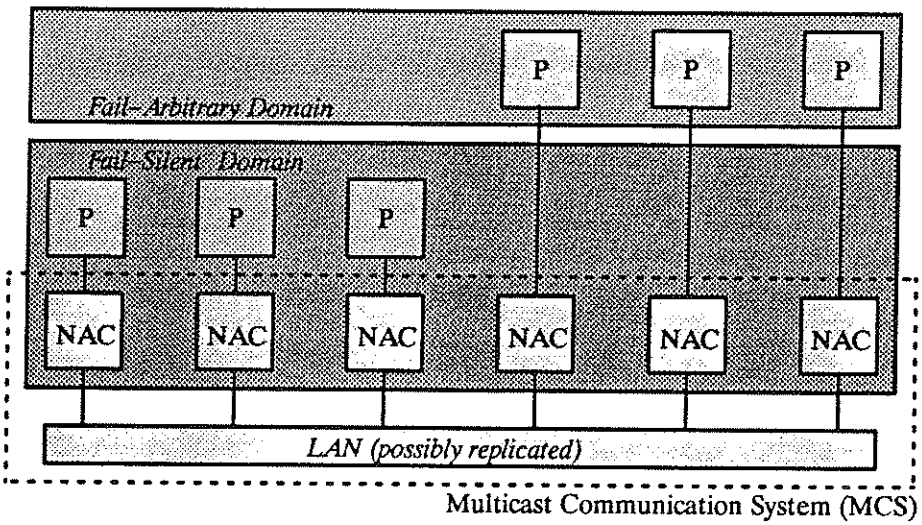

Figure 22 Delta–4 Architecture

The Multicast Communication System (MCS), which consists of the dependable LAN and the *fail–silent* NACs, provides an atomic broadcast mechanism (Atomic Multicast Protocol, AMp [48][49]) which ensures that either all destination processors receive a given message or none do, and has the same ***atomicity***, ***termination*** and ***order*** properties as the atomic broadcast protocols used to implement the *order process* discussed in

Chapter 5. So the algorithms of Chapter 6 can be made to run on top of MCS. The advantage of our approach is that it does not require special hardware such as the NACs.

Delta–4 assumes two classes of processor failures; *fail–arbitrary* and *fail–silent*.

3.10.1 *Active Replication*

A process is replicated and each replica located on a distinct processor in the *fail–arbitrary* domain of Figure 22. Any group of processors may be used, but they must be homogeneous. All replicas execute the algorithm of the process.

When one process group (the source group) sends a message to some other process group (the destination group), the NACs of the two groups execute a two stage algorithm to achieve fault masking. Stage 1 is executed by the NACs of the source process group and serves to arrive at a majority decision. Stage 2 serves to communicate this decision to the NACs (and hence the process replicas) of the destination group.

It was stated in Section 3.1 that a majority decision may be achieved in one stage using oral messages with $2m + 1$ replicas (assuming state divergence is not possible). However, it was assumed that processors communicate directly. In Delta–4, processors communicate through a shared medium, so additional steps are necessary if a majority is to be reached in one stage.

The fail–silent nature of the NACs and the dependable communication medium together ensure that a faulty–processor cannot send different values for a given message to other processors. So the scenario of the inconsistent traitorous commander general of Figure 2 in Section 2.4.3 cannot occur in the Delta–4 system. But because the medium is shared, a faulty processor could assume the identity of another, send a second message and fool the other processors into adopting a majority based on its

incorrect message. To prevent this happening, each processor (or NAC) must sign its message such that any NAC may determine the source (see Section 3.1).

Having achieved a majority, a source NAC may send the single-signed message to the destination group. The fail-silent nature of the NACs and the dependable communication medium together ensure that the same message will arrive at all destination NACs uncorrupted. A destination NAC may therefore pass such a message to its destination process replica without any further voting.

The protocol is now described in more detail:

Suppose that a process P_a wishes to send a message to some other process P_b. Each replica of P_a independently sends a copy of the message to its NAC. If the NAC is not faulty, it *signs* the message and broadcasts it to the other members of the P_a group. If the NAC is faulty, then it will *fail-silent*, and no message will be broadcast. Within the group only a minority of NACs may fail so the majority of NACs will broadcast a message of some kind.

If the message is broadcast, all the NACs in the group P_a will receive it uncorrupted because the LAN is assumed to be dependable. However, the processor executing a replica of P_a may *fail-arbitrary* so there is no guarantee that this message is correct. A NAC must therefore await a majority of authentic signed copies which agree before it may assume that it has received the correct message.

Thereafter, if the NAC is not faulty, it broadcasts this message to all the members of the destination process group P_b. If the NAC is faulty, then it will *fail-silent*, and no message will be broadcast (another NAC in the group P_a will broadcast the message instead). To eliminate the broadcast of unnecessary messages the NACs ensure that only one copy is sent.

Since the LAN is dependable, all the NACs in the group P_b will receive the same message uncorrupted. When a NAC receives a single message, it may assume that it

has received the correct message. The NAC then sends the message to the process replica, P_b.

During stage 2, a single message is sent by a source group NAC to the destination group NACs. The fail-silent nature of the NACs and the dependable communication medium together ensure that a given message is either received correctly by all destination group NACs or none of them (not broadcast). So the broadcast has the ***atomicity*** property of an atomic broadcast. If the message is broadcast it will be received by the destination group NACs within a finite time. So the broadcast also has the ***termination*** property of an atomic broadcast. Because the medium is sequential (it can only carry one message at a time), messages will arrive at the destination group NACs in an identical order. So the broadcast also has the ***order*** property of an atomic broadcast. The stage 2 broadcast thus constitutes an atomic broadcast and ensures that the destination group process replicas receive identical messages in an identical order. Since the destination process replicas are deterministic (they conform to the state machine model referred to in Chapter 1), they will process identical messages in an identical order and state divergence cannot occur.

3.10.2 *Passive Replication*

A process is replicated and each replica located on a separate processor in the *fail-silent* domain of Figure 22. Any group of processors may be used, but they must be homogeneous.

The principles behind passive replication are as follows. One process replica becomes the *active* replica, while the others become the *passive* replicas. Only the *active* replica executes the algorithm of the process. Each time the *active* replica discloses its state to the rest of the system by sending a message, it *checkpoints* that state to the *passive* replicas. If an *active* replica fails, one of the *passive* replicas becomes *active*. The new *active* replica then resumes execution of the process algorithm from the last *checkpoint*

it received as a *passive* replica. In the Delta–4 passive replication model, a process may be *non–deterministic*. Such a new *active* replica could adopt a different execution path to that of its predecessor. But as its predecessor did not produce any output since the last *checkpoint*, no process could have been affected by it.

Because of the fail–silent assumption, the output messages of the *active* replica may be trusted. The *active* replica of the destination process group may therefore accept any messages it receives without voting.

Because all the *passive* replicas change their state according to the *checkpoints* they receive from the *active* replica, there is no possibility of *state divergence*. The processing model may even incorporate pre–emption, something which is not possible in active replication without additional mechanisms. This is because pre–emption in active replication could occur at a different point of execution (and hence a different internal state) in each replica. Actions of the pre–empted process replica which depended on its current state could therefore diverge. In passive replication, only the active replica is pre–empted. If a failure occurs after pre–emption but before a further checkpoint is sent to the passive replicas, the new active replica, when elected, must also be pre–empted.

Delta–4 overcomes many of the disadvantages of other systems, multitasking is allowed, clock synchronization is not required and non–determinism is allowed within passively replicated processes. The disadvantage is that it requires special hardware; the fail–silent NACs and a reliable LAN. In addition, the *fail–silent* NAC may have to be realized using *fail–arbitrary* components (see Section 4.6), in which some form of internal clock synchronization will almost certainly be necessary.

3.11 Delta–4 XPA

Both of the Delta–4 OSA models have advantages and disadvantages. Active replication suffers no time penalty in the case of a failure but incurs an overall time penalty due to the need for authentication and voting protocols. Passive replication minimizes processing requirements and allows non–deterministic processing but suffers a delay in the case of failure. The Delta–4 Extra Performance Architecture (XPA) [50][51][52] introduces a new model, the *leader–follower* model, which seeks to incorporate the advantages of both the models of Delta–4 OSA. This architecture does not require all of the functionality of the atomic multicast protocol (AMp [48]), rather the limited functionality of reliable delivery: either all or none of the functioning receivers receive the message.

In the *leader–follower* model, a process is replicated and each replica located on a different processor in the fail–silent domain of Figure 22. All replicas are *active*, receive input messages and execute the algorithm of the process. Only one replica, designated the *leader*, produces output messages. Since it is assumed that the *leader* is *fail–silent*, its output messages may be *trusted* and voting is unnecessary.

In the OSA active model, non–deterministic processing and pre–emption are prohibited as they could cause *state–divergence* among the replicas. The Delta–4 XPA model incorporates both non–deterministic processing and pre–emption and prevents state–divergence by using synchronizing messages.

All replicas must process their input messages in the same order. Whenever the leader inputs a message, it sends a synchronizing message to its followers containing the identity of that input message. The followers then input that message. Thus the followers always input messages in a consistent order as dictated by their leader.

This mechanism synchronizes replicas at points of input and so may be used to implement pre–emption. All replicas must be pre–empted at the same point in their

execution (to avoid state divergence as referred to in Section 3.10.2). Whenever the leader wishes to input a message, it first checks to see if it should be pre-empted (interrupts etc.). If so, it sends a *pre-empt* message to its followers. The followers then pre-empt at the same point in their execution (the same input statement). The disadvantage of this method is that a process may be pre-empted only when it tries to perform input. Thus the response time to a pre-emption request is dependent on the granularity of input statements in the process. Delta-4 XPA overcomes this disadvantage by explicitly inserting additional pre-emption points into the process code.

At every pre-emption point, the leader checks to see if it should be pre-empted. If not, it increments a *counter* and continues, pre-emption points are thus numbered. If it should be pre-empted, it sends a pre-empt message to its followers containing the current value of its *counter* (a *timestamp*).

When a follower receives a pre-empt message from its leader, a note is made of the *timestamp* contained in the message but no other immediate action is taken. At every pre-emption point a follower also increments a *counter*. If a pre-empt message has been received, the *counter* is compared with the message *timestamp*. If the *counter* has not yet reached the value of the *timestamp*, execution continues. Eventually a pre-emption point is reached where the *counter* equals the value of the *timestamp*, whereupon the follower is pre-empted. The leader and followers thus pre-empt at exactly the same point in their execution.

There is therefore a need for followers to lag behind their leader by at least one pre-emption point. If there was no such time lag, a message sent by the leader, containing a *timestamp* value of n, could be received by a follower after execution of its nth pre-emption point. The leader would thus be pre-empted at n while the follower would be pre-empted at $n+1$, introducing the possibility of state divergence.

3.12 Summary

This chapter has presented several existing systems which use a variety of techniques to prevent state divergence.

In one group (Tandem, Delta–4 OSA passive model), the assumption is made that processors have *fail–silent* semantics, allowing passive replication techniques to be used, so non–determinism within process replicas cannot cause state divergence. Such an assumption is hard to justify in computer systems intended for mission–critical and life–critical applications.

In a second group (SIFT, Mars, Delta–4 OSA active model), state divergence is prevented by constraining process replicas to be deterministic. This prohibits the incorporation of many of the mechanisms found in dynamic real–time systems.

In a third group (FTMP, FTP, Sequoia, Stratus), special hardware is used to achieve tight clock synchronization. Potential sources of non–determinism, such as asynchronous events, are then synchronized to this clock (by hardware) to prevent state divergence.

In a fourth group (Delta–4 XPA, MAFT), agreement protocols are executed to resolve non–determinism and prevent state divergence.

The Delta–4 XPA and MAFT systems are similar to that proposed in this thesis. However, Delta–4 XPA makes the assumption that processors have *fail–silent* semantics, while MAFT may only incorporate non–determinism in process scheduling; processes themselves must be deterministic.

A new architecture will now be presented which seeks to provide an enhanced NMR processing model without the need for specialized hardware or restricted fault assumptions. Processors are grouped to form NMR processing nodes. Protocols are employed within the nodes both to mask faults and to prevent state divergence. Unlike

any of the systems presented in this chapter, the interconnection of nodes is not restricted to any particular architecture or communication medium.

A degenerate form of the NMR node, which has *fail–silent* semantics and supports processes which conform to the enhanced processing model, will then be presented. The protocols used are the same as those used to implement NMR nodes and, as with the NMR architecture, the interconnection of nodes is not restricted to any particular architecture or communication medium.

Chapter 4 : System Architecture

4.1 Introduction

This chapter considers the application of NMR techniques to a distributed system of processors. As stated in Section 3.12, processors are grouped to form NMR processing nodes but the interconnection of nodes is not restricted to any particular architecture or communication medium. In this chapter however we shall consider the specific case of a collection of TMR nodes connected via point–to–point communication links.

It is intended to achieve replication transparency. That is, any problems posed by replication and voting are hidden from the application programmer who is then only concerned with the development of a non–redundant system. The architecture is aimed primarily at compute–intensive real–time applications and is a practical one. Microprocessors such as the Inmos Transputer [65] were designed specifically to enable construction of such networks with little hardware overhead (at present, a Transputer has 4 bi–directional 10Mbit/sec serial communication links).

The chapter is structured as follows. Section 4.2 develops a processing model and shows how this may be replicated using TMR techniques. Section 4.3 presents a processor interconnection scheme to support the replicated processing model which remains fully connected after one processor failure per TMR node (triad). Section 4.4 describes the signature and authentication mechanisms used in voting while Section 4.5 presents the algorithm for voting and shows how duplicate messages may be detected and discarded. Section 4.6 describes the properties of a degenerate form of our system in which process and processor triads are replaced by process and processor pairs, while Section 4.7 summarizes the contributions of this chapter.

4.2 The Replicated Processing Model

We assume that application programs can be mapped on to a number of processes that interact via messages. Communicating processes have bi-directional links between them. Figure 23(a) shows a system of five concurrent processes with six links. A link connecting two processes has the property that the messages sent by one process are received by the other process uncorrupted and in the sent order. We also assume that if a process with multiple links (such as c_2 in Figure 23(a)) simultaneously receives messages on those links then these messages are chosen non-deterministically for processing. Message selection is however assumed to be fair, that is, the process will eventually select a message present on a link.

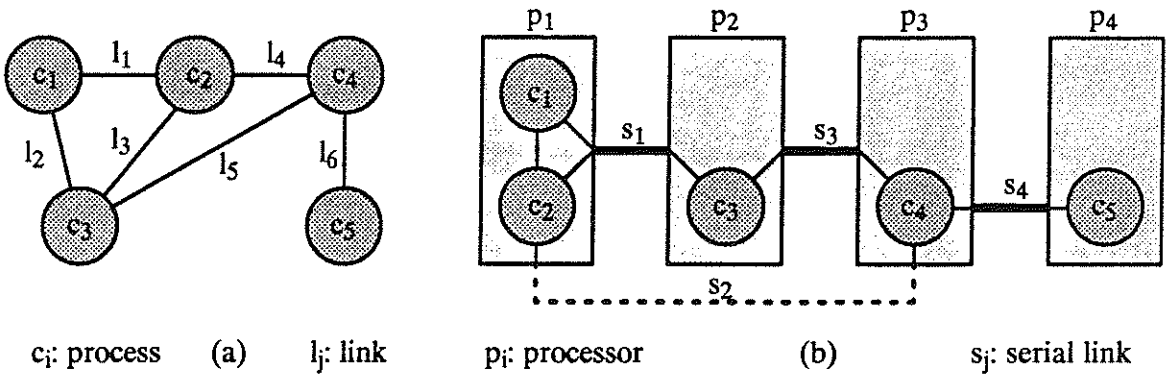

Figure 23 Process–Processor Mapping

We further assume that such a system of processes can be configured to run on a given set of processors, each possessing a number of bi-directional communication links, as shown in Figure 23(b) for example. Each process is mapped onto a physical processor (eg. c_3 mapped onto p_2). This may require *multitasking* (eg. c_1 and c_2 mapped onto p_1). Each inter-process link is mapped onto a physical link (eg. l_5 onto s_3). This may require the *multiplexing* of several links onto one physical link (eg. l_2, l_3 mapped onto s_1). If such a process to processor mapping places two communicating processes on processors which are not directly connected (eg. s_2 does not exist), then the intermediate

processors must have the capability of relaying messages (*through–routing*). If replication transparency is to be achieved, *multitasking*, *multiplexing* and *through–routing* must be invisible to the processes themselves.

We now consider the problem of making a system of concurrent processes tolerant to a bounded number of processor failures. Given a non–redundant system of C ($C \geq 1$) concurrent processes partitioned to run on P ($P \leq C$) number of processors, we address the problem of constructing a voted replicated system of N*C processes ($N = 2m + 1$, $m \geq 1$) partitioned to run on N*P processors and capable of tolerating up to P*m processor failures. Each process $c \epsilon C$ is replaced by a group of processes with N members, and each processor $p \epsilon P$ is replaced by a group of N processors.

From now on we will assume the degree of replication to be three giving us the well known Triple Modular Redundant (TMR) system.

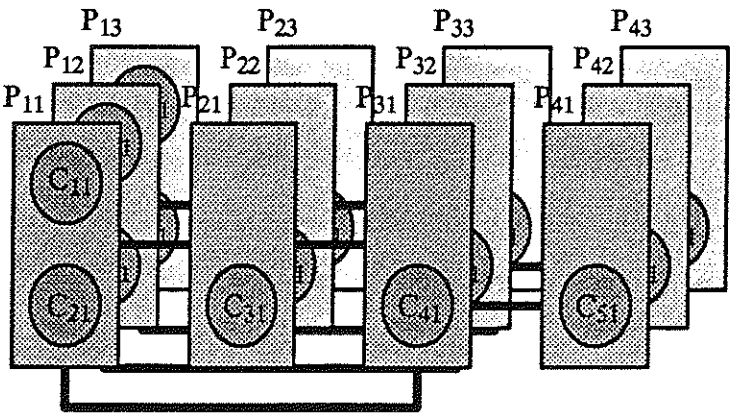

Figure 24 Replicated System

In the replicated version of the system of Figure 23, shown in Figure 24, each process c_i ($1 \leq i \leq 5$) is replaced by a process triad C_i, such that $C_i \equiv \{C_{i1}, C_{i2}, C_{i3}\}$ and each processor p_j ($1 \leq j \leq 4$) is replaced by a processor triad P_j such that $P_j \equiv \{P_{j1}, P_{j2}, P_{j3}\}$ with C_{i1} mapped onto P_{j1}, C_{i2} onto P_{j2} and C_{i3} onto P_{j3}.

A processor with its links will be treated as a single entity, so a processor with either faulty links or a faulty processing unit or both will be treated as a faulty processor. A failed processor may behave in an arbitrary manner and hence, processes of a failed processor may behave in an arbitrary manner. Assuming that at most one processor in each triad may fail, at most one process in each process triad may be faulty.

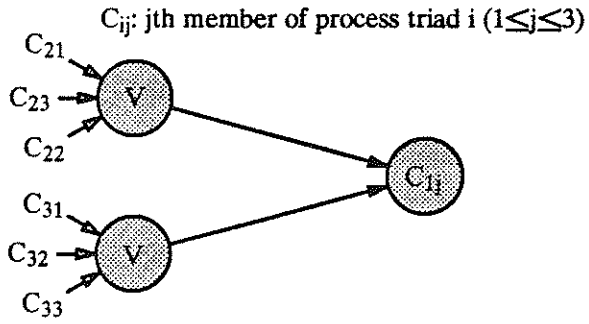

Figure 25 Voting of Incoming Messages

A non-faulty process must reject inputs from a faulty process; this is achieved by majority voting as shown in Figure 25 which depicts the voters for the jth member of process triad C_1 (The replicated version of process c_1 in Figure 23(a)).

The function of each process c_i, is to repeatedly pick up an input message from one of its links and perform some action, as shown in Figure 26.

```
process ci:
    var m:message
cycle
    receiveany(m)
    action(m)
end
```

Figure 26 Unspecified Input

If a process with multiple links has input messages pending on more than one link, then one is selected *non-deterministically*. Message selection is however assumed to be *fair*, that is, every message will eventually be selected. The computation performed by a

process (action(m)) on receipt of a particular selected message is *deterministic.* In particular this means that if all non–faulty process replicas of a triad have identical initial states and process identical messages in an identical order, then identical output messages in an identical order will be produced. The above model conforms to the state machine model presented in Section 2.2 and discussed in [7] where, "Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in the system".

In Figure 25, if C_{1j} receives two voted messages simultaneously then one of them will be chosen non–deterministically for processing. As a consequence, if the replicas were to act unilaterally, they could make a different choice, produce non–identical outputs and thereby defeat the voting mechanism. It is necessary therefore to ensure that each non–faulty replica performs *identical message selection* from the set of available input messages.

The problem of identical message selection may be transformed to that of *identical message ordering* by presenting a single input message queue to a process and ensuring that only the message at the head of the queue may be selected. An *order process* may then employed to ensure *identical message ordering* in the queues as described in Chapter 5.

This is the basis of the replicated state machine approach presented in [7]. Other approaches particularly suited to replicated processing with voting are presented in [8][80]. Some practical systems have opted for more restrictive models (without non–determinism) to avoid the overheads associated with ordering (examples are SIFT [28] and MARS [31]); while other work on replicated distributed programs [60] did not address the issue of concurrent processing and ordering.

The model presented here is sufficiently general in that other models, such as clients and servers interacting through remote procedure calls [61], or objects communicating

by messages can be seen as special cases. However, several important enhancements are possible and will be discussed in Chapter 6.

4.3 Processor Interconnection

Processor interconnection architectures and voting algorithms are closely related. If Byzantine faults are allowed and signatures are not used to protect messages against corruption (*oral* messages) then a message can be *trusted* (non–faulty if the sender is non–faulty) only if it is received directly from the sender. So processors of communicating triads must be fully connected as shown in Figure 27.

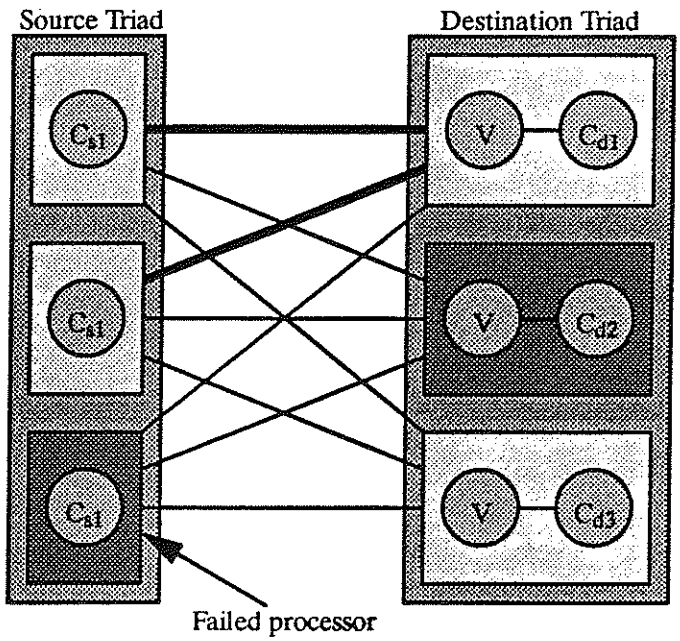

Figure 27 Communicating Triads (oral messages)

If we allow one processor failure per triad, the destination voter of a non faulty processor will receive at least two uncorrupted copies of a message from the source component replicas on non–faulty processors and will be able to form a majority (as shown in bold for C_{d1} in Figure 27).

In a distributed system of triads where one processor failure per triad must be tolerated, either every potentially communicating pair of triads must be fully

connected or every adjacent pair must be fully connected and messages must be voted at all intermediate triads as well as at the destination.

If *signatures* and *authentication* (see *signed* messages in Section 4.4) are available then the architecture may be simplified to that shown in Figure 28. Voting may be carried-out at the source or destination (see Section 4.5) but the architecture remains the same.

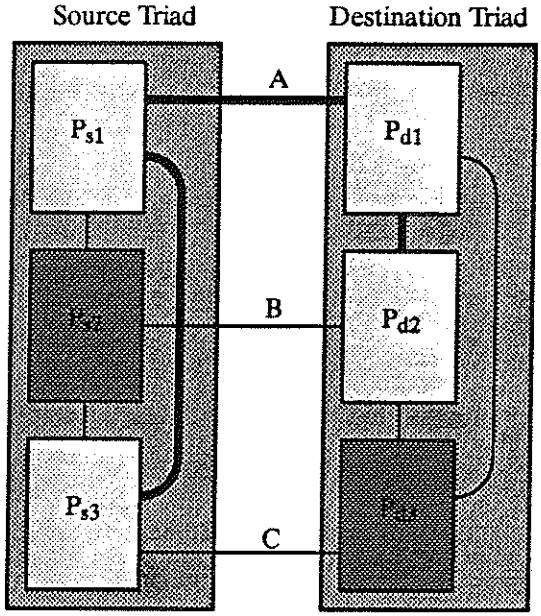

Figure 28 Communicating Triads (signed messages)

If we allow at most one processor failure per triad, every non-faulty processor remains connected to every other non-faulty processor; either directly, or via other non-faulty processors, as shown by the bold lines in Figure 28. Each source component replica sends a copy of it's message to each of it's neighbours within the triad and relays any authentic messages it receives. Therefore at least two non-corrupted copies of every message will be received by every voter, whether at source or destination.

In a distributed system of triads, an intermediate triad processor must also send copies of each message it receives to it's neighbours within the triad. If the potentially large

number of duplicates is to be reduced, they must authenticate every message and relay only the first authentic copy. This problem will be addressed in Section 4.5.4.

Current research [62] into distributed TMR systems assumes that the triads are embedded into one homogeneous communication structure as shown in Figure 29.

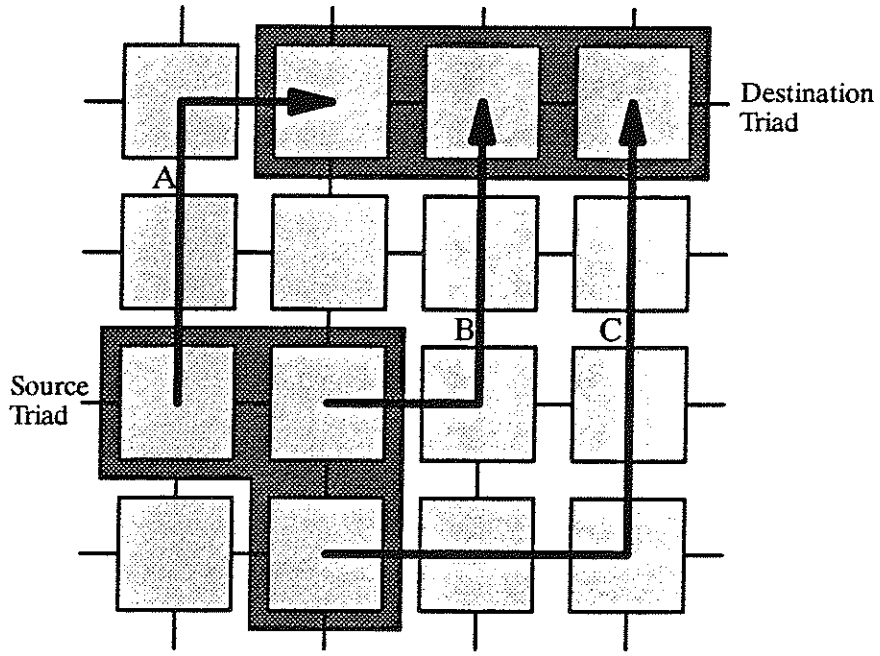

Figure 29 Embedded Triads

This has several disadvantages:

- The routing algorithm must ensure that the three inter–triad communication paths A, B and C (shown in bold in Figure 29) and the six intra–triad communication paths (not shown) remain distinct; to prevent the failure of a single intermediate processor corrupting two or more messages.
- Determining the number and position of *tolerable* failures is a complex procedure and depends on the relative positions of the source and destination triads (no concept of intermediate triads and one failure per triad).

Alternatively, paths A, B and C may lie in three totally disjoint communication planes as shown in Figure 30.

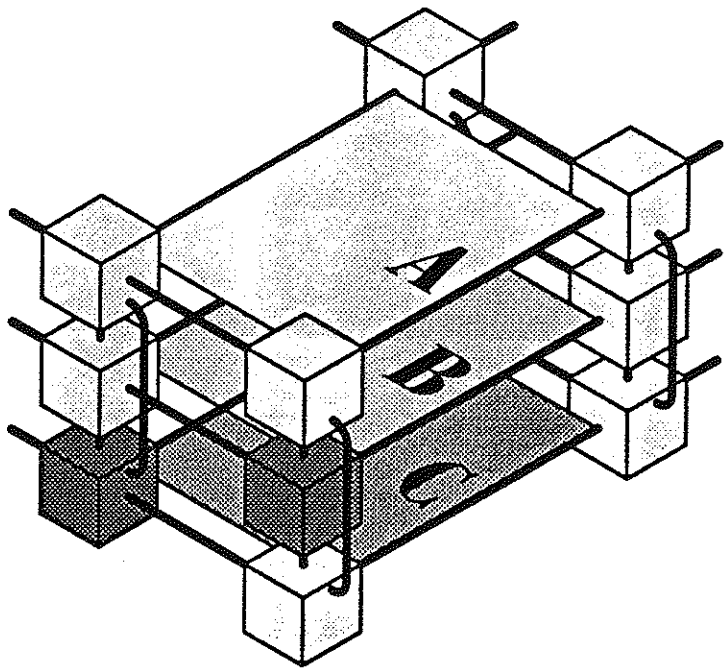

Figure 30 Distinct Communication Planes

Several advantages become apparent:

- A simpler (more efficient and reliable) routing algorithm may take any path to reach it's destination; path isolation is guaranteed.
- Intermediate processors may be grouped into triads to provide symmetry.
- If the routing algorithm is identical for each plane, paths A, B and C will pass through the same set of intermediate triads, simplifying authentication and the passing of messages between planes.
- Processors within a triad are directly connected in a local *voting plane* so voting algorithms become more efficient and do not depend on the correctness of intermediate processors.

We will assume a *three-plane* architecture for the remainder of this thesis.

If a processor has four bidirectional links (as is the case with current Transputers) a pipeline (or a ring) of processor triads may be constructed as shown in Figure 31.

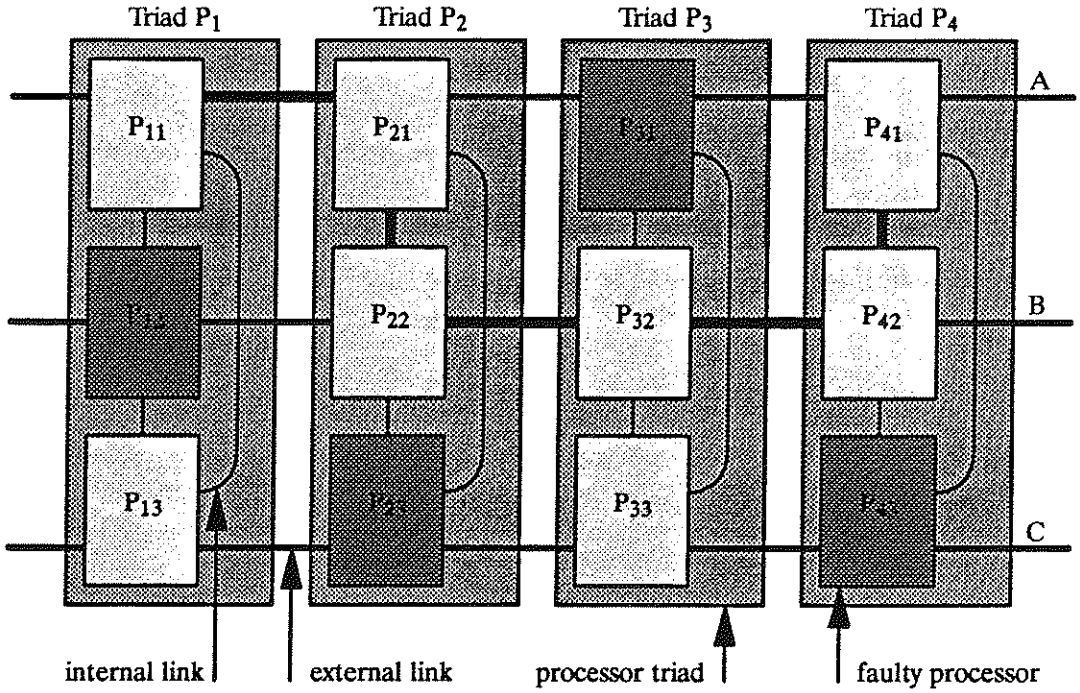

Figure 31 Processor Interconnections

We will refer to a link connecting processors of a triad (within the voting plane) as internal and a link connecting processors of two adjacent triads (within a communication plane) as external.

The pipelined nature of interconnection means that a message from one process to some other process may have to be relayed by intermediate processors. From a TMR system we require:

M1 the capability of masking at most one processor failure per triad.

Under such a failure assumption, the pipelined structure possesses the following two connectivity properties:

C1 Any non–faulty processor of a triad is remains directly connected to all other non–faulty processors of the same triad.

C2 From any non–faulty processor in a triad P_i, there is a non–faulty path connecting that processor to any non–faulty processor of any other triad P_k.

A path between any two non–faulty processors is non–faulty if all the intermediate processors in the path are non–faulty (for example, the path between P_{11} and P_{41} drawn in bold lines in Figure 31 is non–faulty).

Any architecture which possesses *C1* and *C2* may support a voting mechanism which implements *M1* (see Section 4.5). Hence the pipeline architecture is a suitable candidate for replication.

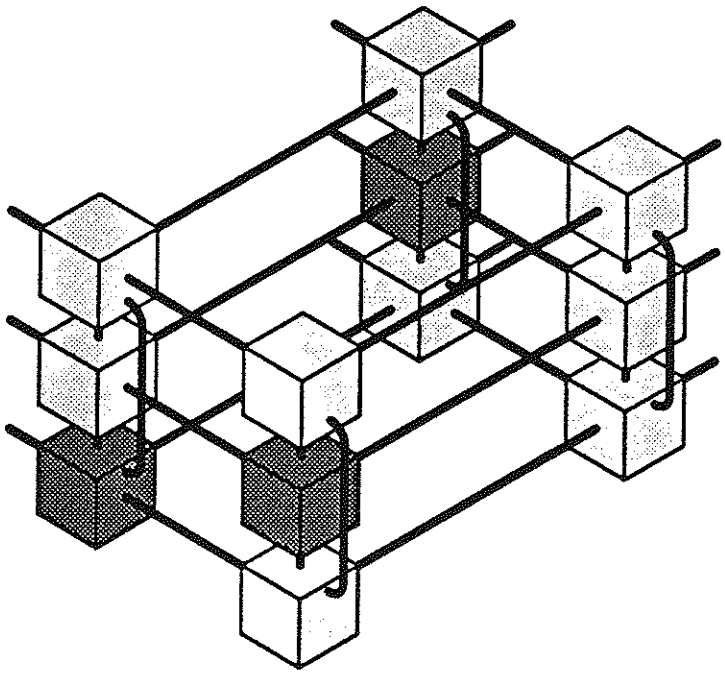

Figure 32 A Triplicated Grid

If more than four links per processor are available, then structures other than a pipeline can be formed. For example, Figure 32 shows a small section of a triplicated

grid structure which requires up to six links per processor (a six link processor could be built by *clustering* two four link processors).

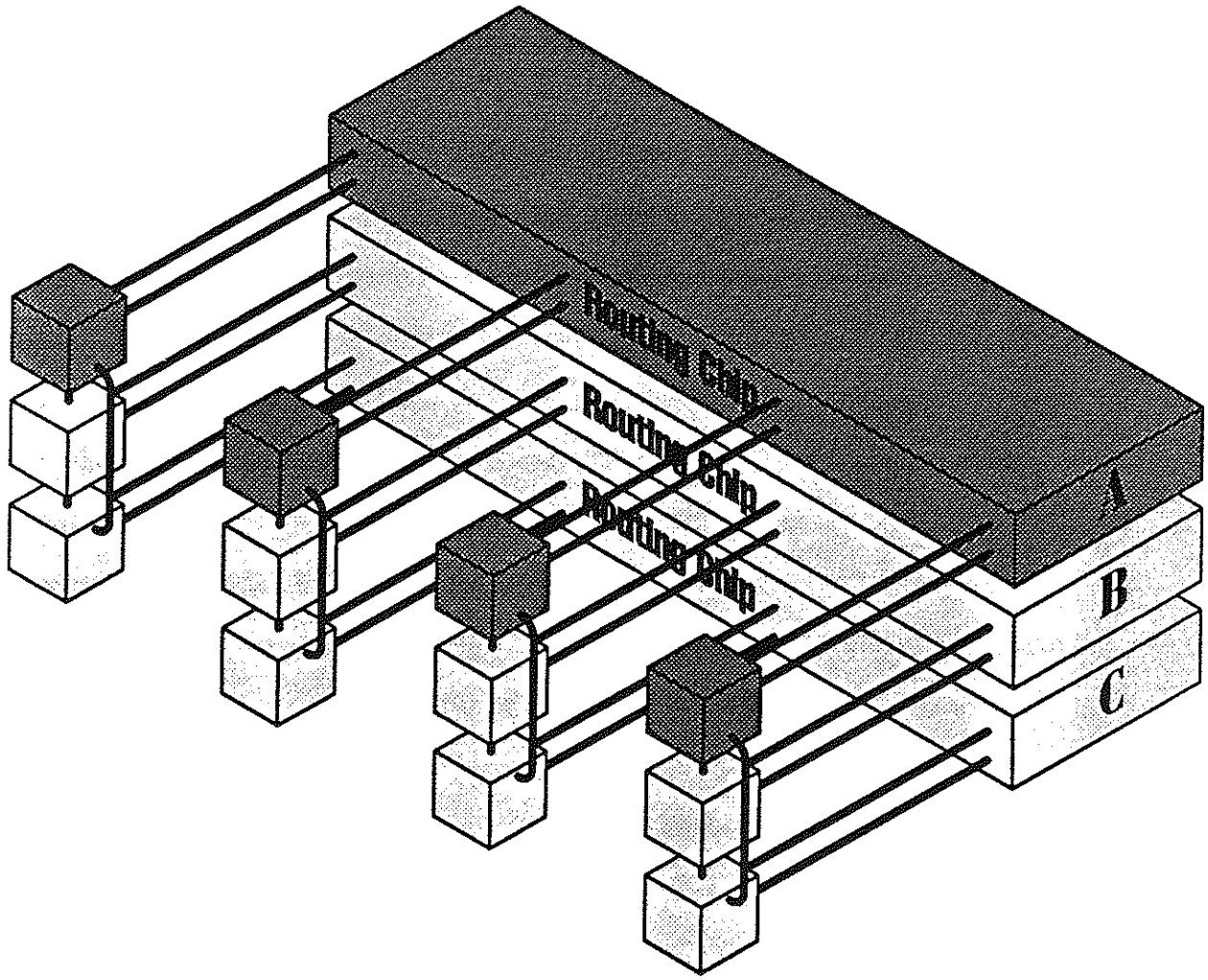

Figure 33 Communication using Routing Chips

The grid structure also possesses *C1* and *C2*. In fact, any structure in which the P_{i1}'s, the P_{i2}'s and the P_{i3}'s are independently connected by an identical network possesses *C1* and *C2*. This means that any non-replicated structure (bus, grid, hypercube etc.) has an equivalent replicated counterpart which satisfies *M1*. Such a replicated structure is constructed by transforming each processor p_i into a triad P_i and connecting P_{i1}, P_{i2} and P_{i3} directly.

Interprocessor communication speeds in the grid structure of Figure 32 depend on the *distance* between the communicating parties and the processing load on intermediate processors (through–routing is not automatic but requires the cooperation of the relaying processor). Figure 33 shows the replication of a structure envisaged by the designers of the new Transputer family (at present known only as *the H1*).

The management of interprocessor communication is delegated to a purpose–built routing chip (or network of such chips) to increase the speed of interprocessor communication. Each Processor has one or more connections to a routing chip. This architecture has an inherent weakness in fault–tolerance terms. Failure of a single routing chip would incapacitate a large number of processors (up to one third as shown in Figure 33). Although the voting mechanism would tolerate such a failure, a single fault arising in any of the remaining processors would cause system failure. The problem may be overcome to some extent by increasing the number of routing chips and connecting each processor to more than one of them. But in the limit, to achieve the same level of fault–tolerance as in the grid structure of Figure 32, one routing chip per processor would be necessary. Failure of a routing chip would be then considered as a failure of the attached processor.

4.4 Signatures and Authentication

The algorithms in this thesis refer to signed and double–signed messages. Both may be generated using a *public–key cryptosystem* mechanism (based on that published in [63]) which will now be described.

Processor P_i *encrypts* a message M using an encryption procedure E_i and *decrypts* a message using a decryption procedure D_i. A procedure consists of a *general method* (G_E or G_D) common to all processors and a processor–specific *key* (K_{Ei} or K_{Di}). The encryption key K_{Ei} is made public and is therefore known to all processors which wish

to communicate with P_i, but the decryption key K_{Di} is kept secret and is therefore known only to P_i. The procedures have the following properties.

- Both $E_i(M)$ and $D_i(M)$ are easy to compute if K_{Ei} and K_{Di} are known.
- There is no efficient method of computing $D_i(M)$ without knowing K_{Di}.
- $D_i(E_i(M)) = M$
- $E_i(D_i(M)) = M$

4.4.1 *Single-signed Messages*

To generate a single-signed message, we require that:

SIG1 A non-faulty processor must be able to *sign* a message such that the non-faulty recipient of that message can verify it's *correctness* (message received = message sent) and determine the *signer-identity* (collectively termed *authentication*).

Suppose that a message M_0 is to be sent by some processor P_j to some other processor P_k.

A new message M_1 is constructed by appending to M_0 the identity of P_j (I_j) and any other relevant information (O_j) such as the *destination-identity* (I_k) or a *sequence number* (see Section 4.5.4).

$$M_1 = (M_0 + I_j + O_j)$$

A *signature* S_j is calculated by decrypting M_1.

$$S_j = D_j(M_1) \qquad [= D_j(M_0 + I_j + O_j)]$$

A new message M_2 is constructed by appending this signature to M_1 and sent to P_k.

$$M_2 = (M_1 + S_j) \qquad [= (M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j)]$$

When M_2 is received by P_k, the identity of the signer (I_j) is extracted and used to retrieve the appropriate encryption function E_j. The signature S_j is then extracted and encrypted using E_j to reveal the purported original message M_1'.

$$M_1' = Ej(Sj) \qquad [= E_j(D_j(M_0 + I_j + O_j))]$$

$$= (M_0 + I_j + O_j) \ ?$$

M_1 is then extracted and compared with M_1'. If they are equal, then the message M_2 (and hence M_0, I_j and O_j) must have been received uncorrupted (with high probability) and M_1 must have been decrypted using E_j. So the original message must be M_0, and it must have been signed by P_j.

4.4.2 *Double–signed Messages*

To generate a double–signed message, we require that:

SIG2 On receipt of a single–signed message, a non–faulty processor must be able to extract and compare the original message (M_0) with one of it's own making and if equal, be able to *countersign* the message such that the non–faulty recipient of that message can verify it's *correctness* and determine both the *signer–identity* and the *countersigner–identity* (collectively termed *authentication*).

Suppose that a message M_2 has been sent by some processor P_j to some other processor P_k as described in Section 4.4.1, and that the message must be relayed to some third processor P_l. The message received by P_k is:

$$M_2 = (M_1 + S_j) \qquad [= (M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j)]$$

A new message M_3 is constructed by appending to M_2 the identity of P_k (I_k) and any other relevant information (O_k).

$$M_3 = (M_2 + I_k + O_k)$$

$$[= (M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k]$$

A *signature* S_k is calculated by decrypting M_3.

$$S_k = D_k(M_3)$$

$$[= D_k((M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k)]$$

A new message M_4 is constructed by appending this signature to M_3 and sent to P_l.

$$M_4 = (M_3 + S_k)$$

$$[= (M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k$$

$$+ D_k((M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k)]$$

When M_4 is received by P_l, the identity of the countersigner (I_k) is extracted and used to retrieve the appropriate encryption function E_k. The signature S_k is then extracted and encrypted using E_k to reveal the purported original message M_3'.

$$M_3' = E_k(S_k)$$

$$[= E_k(D_k((M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k))]$$

$$= (M_0 + I_j + O_j) + D_j(M_0 + I_j + O_j) + I_k + O_k \ ?$$

M_3 is then extracted and compared with M_3'. If they are equal, then the message M_4 (and hence M_3, I_k and O_k) must have been received uncorrupted (with high probability) and M_3 must have been decrypted using E_k. So the intermediate message must be M_2, and it must have been countersigned by P_k.

M_2 is now processed by P_l (as it was processed by P_k in Section 4.4.1). That is, the identity of the signer (I_j) is extracted and used to retrieve the appropriate encryption function E_j. The signature S_j is then extracted and encrypted using E_j to reveal the purported original message M_1'.

$$M_1' = E_j(S_j) \qquad [= E_j(D_j(M_0 + I_j + O_j))]$$

$$= (M_0 + I_j + O_j) \ ?$$

M_1 is then extracted and compared with M_1'. If they are equal, then the message M_2 (and hence M_0, I_j and O_j) must have been received uncorrupted (with high probability)

and M_1 must have been decrypted using E_j. So the original message must be M_0, and it must have been signed by P_j.

To summarize, the original message must be M_0, and it must have been signed by P_j and countersigned by P_k.

4.5 Voting and Authentication

The voting algorithm must allow the sending of messages between one triad (source) and another (destination) in the presence of up to one processor failure per triad. The problem may be broken-down into the three sub-problems of how to send the message of each replica in the source processor to all replicas in the destination triad. This is another manifestation of the *Byzantine Generals Problem* of Section 2.4.2. However, whereas in the *Byzantine Generals Problem* we must satisfy *IC1* even in the case where the commanding general is traitorous, in the voting algorithms presented here we need make no such assumption (see Section 3.1).

Consider the architecture of Figure 34 where non-faulty processors remain connected via a non-faulty path in the presence of a single processor failure per triad (shown by the bold lines).

4.5.1 *Single–signed Messages*

Assume that a process triad C_m is mapped onto processor triad P_1, and a process triad C_n is mapped onto processor triad P_4 and that C_m wishes to send a message to C_n. The voting of messages may be achieved as follows. For each $j = 1..3$, the message is signed by P_{1j} and sent to its neighbours P_{1k} ($j \neq k$) and to it's successor P_{2j}. Any message received by P_{1j} from it's neighbours is also sent on to P_{2j}. Any message received by P_{2j} from P_{1j} is sent to it's neighbours P_{2k} ($j \neq k$) and sent on to it's successor P_{3j}. Any

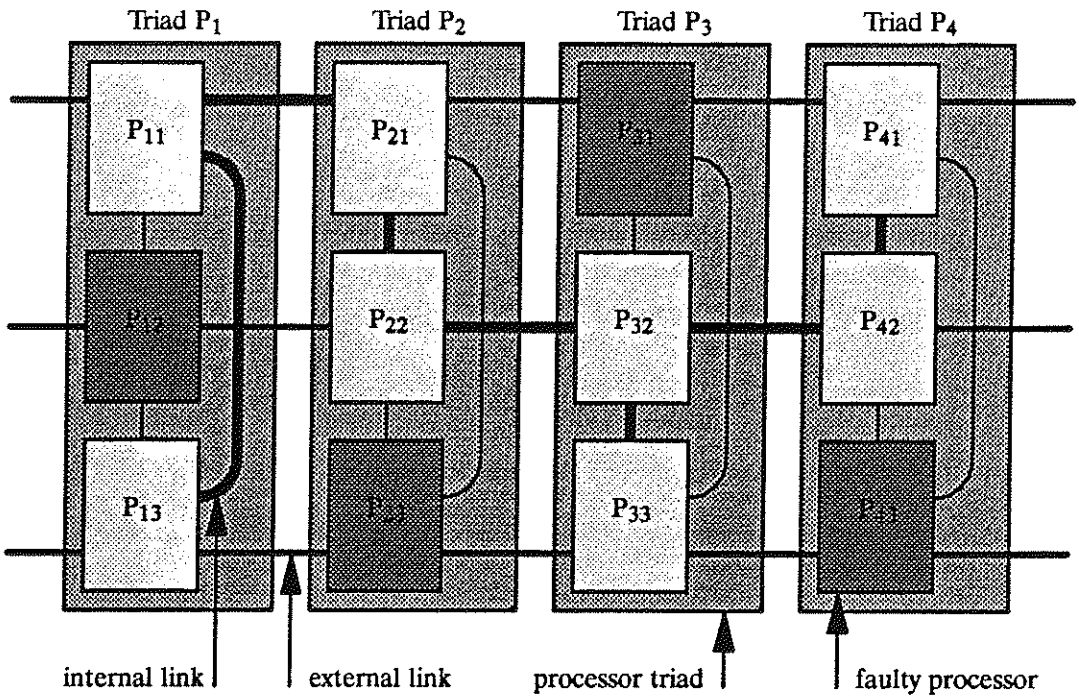

Figure 34 Processor Interconnections

message received by P_{2j} from a neighbour P_{2k} ($j \neq k$) is sent on to it's successor P_{3j} (this overcomes the case of a faulty P_{1j}). P_{3j} then behaves as P_{2j}. When the message is received by the destination processor P_{4j}, it is sent to the neighbouring processors P_{4k} ($j \neq k$) and passed to the destination process c_{nj}. Any message received by P_{4j} from a neighbour P_{4k} ($j \neq k$) is also passed to the destination process c_{nj} (this overcomes the case of a faulty P_{3j}).

Every inter–triad message is signed by it's originator P_{1j}. Since we assume at most one processor failure per triad and that a signature cannot successfully be forged, when the destination processor P_{4j} receives two authentic matching copies from two different originators, at least one of the copies must be from a correct source processor and therefore both copies must be correct.

One disadvantage of this method (voting at destination) is the unnecessary generation of a large number of duplicates messages. In Figure 34, if there were no faulty processors, a total of eighty–one messages would pass between P_{3j} and P_{4j}. In the presence of faults an unlimited number of additional authentic messages could be injected into the system by a *babbling* processor. Additional steps are necessary to limit the number of inter–triad messages.

For each message, the destination processor P_{4j} needs only two authentic matching copies from two different originators for successful voting to take place. Since there is always at least one non–faulty path between each non–faulty source processor and each non–faulty destination processor, only two messages need be sent along each link. Therefore P_{1j} need only send two messages to it's successor P_{2j}; it's own and the first authentic one from a neighbour which matches. P_{2j} need only send two messages to it's neighbours P_{2k} ($j \neq k$) and it's successor P_{3j}; the first two authentic messages from a different originator. P_{3j} then behaves as P_{2j}. Finally, P_{4j} need only send two messages to it's neighbours P_{4k} ($j \neq k$); the first two authentic messages from a different originator. The number of messages passing between P_{3j} and P_{4j} (or any other pair of adjacent triads) has now been reduced to six.

4.5.2 *Double–signed Messages*

To reduce the number of inter triad messages to six as described in Section 4.5.1, P_{1j} must authenticate a message received from a neighbour and compare it with it's own message before sending it on. Authentication and voting is therefore taking place at the source triad as well as at the destination triad. Suppose that P_{1j} were to append it's own signature to authentic messages received from it's neighbours before sending them on (voting at source). An inter–triad message is now signed by two members of the source triad. Since we assume at most one failure per triad, the message must have been seen and signed by at least one non–faulty processor. If the message arrives at the destination uncorrupted, it must therefore be correct (with high probability). This fact allows the destination triad to continue processing after the receipt of a single authentic double–signed message, without any further voting. So processors need only send one authentic double–signed message to their successor ($P_{1j}..P_{3j}$) and neighbours ($P_{2j}..P_{4j}$). The number of messages passing between P_{3j} and P_{4j}. (or any other pair of adjacent triads) has now been reduced to three.

The algorithm will now be considered in more detail. The mechanisms for the removal of duplicates have been omitted for clarity; discussion will be deferred until Section 4.5.3.

Referring to Figure 35, a processor maintains four message pools :

- **Processed Message Pool (PMP)** : Contains unsigned messages produced by local process triads (C_{ij}s).
- **Received Message Pool (RMP)** : Contains single-signed messages that are received from neighbours and found to be authentic.
- **Candidate Message Pool (CMP)** : Contains locally produced unsigned messages waiting for a signed message of identical contents to arrive at RMP.
- **Voted Message Pool (VMP)** : Contains voted messages (stripped of their double-signatures) intended for members of local process triads (C_{ij}'s).

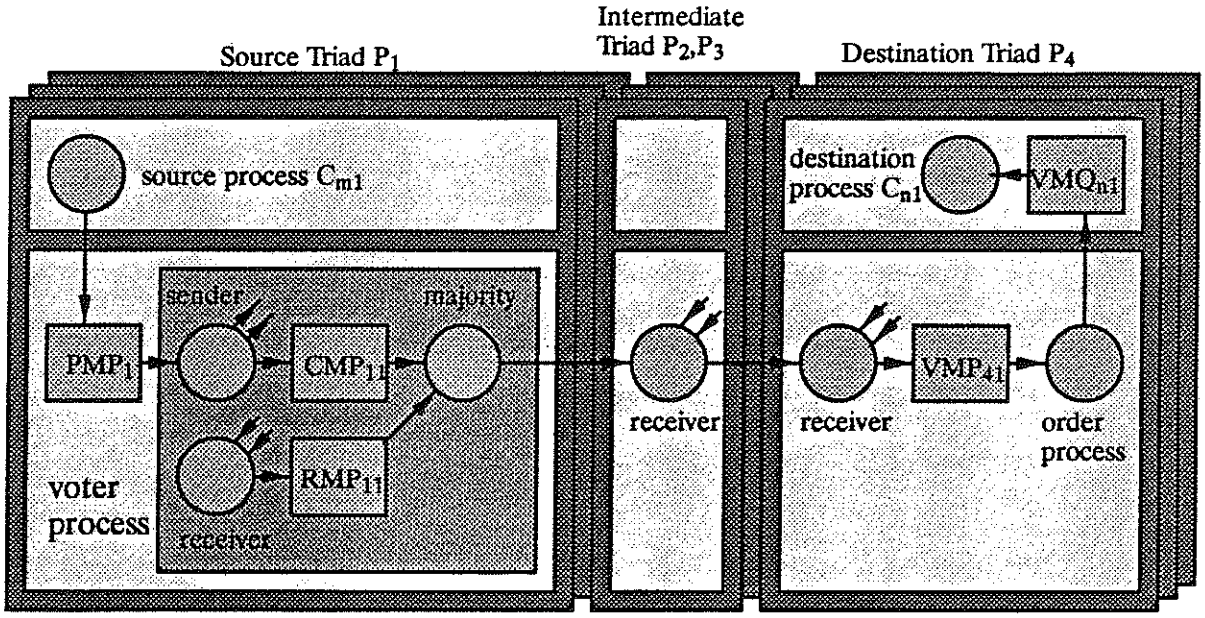

Figure 35 Message Flow

In addition, each process C_{ij} has a queue (VMQ_{ij}) containing messages to be processed. As we shall see in subsequent chapters, the order of messages within VMQ_{ij} is ensured by an order process to be identical among members of the process triad.

```
process voter:
    process sender:
        var m:message
        cycle
            remove(PMP, m)
            deposit(CMP, m)
            send_to_neighbours(signed(m))
        end
    end
    process receiver:
        var m:message
        cycle
            receive_from_neighbours(m)
            if not authentic(m) then
                discard(m)
            else deposit(RMP, m)
            endif
        end
    end
    process majority:
        var m:message
        cycle
            remove_pair(RMP, CMP, m)
            send_to_successor(signed(m))
        end
    end
end
```

Figure 36 Voter Process Algorithms

The voter process is composed of three concurrent processes: sender, receiver, and majority as shown in Figure 36. The *voter.sender* process selects a message from PMP, places a copy in CMP and sends signed copies to the other members of the source triad. The *voter.receiver* process collects these single–signed messages in RMP. The *voter.majority* process tries to *pair–up* messages from pools CMP and RMP. If a *pair* can be formed, the copy from RMP is countersigned and the resulting double signed message is sent towards the destination.

```
process receiver:
    var m:message
    cycle
        if not authentic(m)
        then discard(m)
        else
            send_to_neighbours(m)
            if m.dest = me then
                deposit(VMP, m)
            else
                send_to_successor(m)
            endif
        endif
    end
end
```

Figure 37 Receiver Process Algorithm

The double–signed message is received at the neighbouring triad by a *receiver* process. If the message is authentic and has not yet reached it's destination (intermediate triad), it is sent to other members of the triad and relayed towards it's destination, as shown in Figure 37. If the message is authentic and has reached it's destination (destination triad), it is sent to other members of the triad and a copy, is deposited in VMP.

The unsigned message is transferred from VMP to VMQ by the order process in such a way that the order of messages in VMQ is identical to that in the other replicas.

When the destination triad is the same as the source, the message need not be voted (a faulty source process implies a faulty destination process). So, as an optimization, the source process may in this case deposit the message directly into the local VMP.

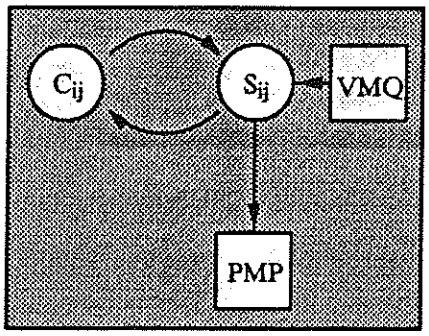

Figure 38 The Stub Process

In Figure 35, c_i has only one outgoing link (to PMP) and one incoming link (from VMQ). This violates the replication transparency requirement. One way in which the requirement may be satisfied is by the addition of a stub process which emulates the original links of c_i as shown in Figure 38.

4.5.4 *Detection of Duplicate Messages*

As they stand, the algorithms in Figure 36 and Figure 37 would generate many duplicates. Also, although corruption of a message would be detected by a *voter.receiver* or *receiver* process and would prevent it's propagation (a faulty processor cannot flood the system with corrupted messages) a faulty processor could repeatedly send the same authentic message, causing RMP or VMP to overflow. For these two reasons, the *voter.receiver* and *receiver* processes must detect a duplicate message and prevent it's propagation.

This could be accomplished by maintaining a history of propagated messages but such a history would be infinite. The history could be made finite if the message were to

carry a timestamp protected by the double–signature and a message older than some predetermined value was deemed a duplicate. But this requires the processors of the source triad to agree on the timestamp (difficult in a Byzantine world) and a faulty processor could still send an arbitrary number of messages during this period.

What is needed is a method whereby a process can determine whether or not a message is a duplicate immediately upon receipt of that message.

It is assumed that a message carries source and destination identifiers and a process can determine if the source (and destination) processes of two independent messages belong to the same triad. If the messages differ in either source process triad or destination process triad they cannot be duplicates. The problem is therefore reduced to that of detecting duplicates for a given source triad/destination triad pair.

It is assumed that communication is synchronous. That is, at most one message (excluding duplicates) will be in transit between a given source triad and destination triad at any point in time. Now suppose that each message is *timestamp*ed by the source process and that duplicates of a message carry the same *timestamp*. A process could detect a duplicate message by comparing it's *timestamp* with that of the last non–duplicate message received. Values equal or lower would signify a duplicate message. A higher value would signify that the message was a *new* one and not a duplicate. Unfortunately, the method, which is a form of *temporal ordering*, requires the synchronization of the clocks of processors within the source triad (difficult in a Byzantine world).

The absolute time that a message was sent is not of importance here, only it's relationship to other messages between the same source triad/destination triad pair. We may establish a *happened before* (or more accurately *happened at the same time*) relationship by employing synchronized *logical clocks* [5] instead of synchronized *real–time clocks*.

Consider the following. Every message carries a sequence number. A message is discarded if it carries a sequence number equal to or lower than that of a message already received (between the same source triad/destination triad pair). In practice

```
process voter:
    process sender:
        var m:message
        var counter[][]:sparse array of Int:=0
        cycle
            remove(PMP, m)
            m.counterstamp:=counter[m.src][m.dest]
            deposit(CMP, m)
            send_to_neighbours(signed(m))
            inc counter[m.src][m.dest]
        end
    end
    process receiver:
        var m:message
        var counter[][]:sparse array of Int:=0
        cycle
            receive_from_neighbours(m)
            if not authentic(m) then discard(m)
            else
                if m.counterstamp=counter[m.src][m.dest] then
                    deposit(RMP, m)
                    inc counter[m.src][m.dest]
                elseif m.counterstamp<counter[m.src][m.dest] then discard(m)
                elseif m.counterstamp>counter[m.src][m.dest] then
                    serious_error(m)
                    discard(m)
            endif
        end
    end
end
```

Figure 39 Modified Voter Process Algorithms

this means that every process on the path from source to destination must maintain a counter unique to that source triad/destination triad pair (*counter[s,d]*). A *counter[s,d]* is initialised to zero when the first message between *s* and *d* is received. In the

discussion that follows, it will be assumed that corrupted messages have already been discarded.

Consider the modified voter process algorithms in Figure 39. When a message is removed from PMP by the *voter.sender* process, it is stamped with the current *voter.sender.counter[s,d]* value and the latter is incremented. When a message is received by the *voter.receiver* process, the *counterstamp* is compared with the *voter.receiver.counter[s,d]* value.

- If *counterstamp* = *voter.receiver.counter[s,d]* the message is placed in RMP and the counter incremented.
- If *counterstamp* < *voter.receiver.counter[s,d]* the message is a duplicate and may be discarded.
- If *counterstamp* > *voter.receiver.counter[s,d]* the message is from a faulty neighbour and may also be discarded.

This mechanism removes the redundant duplicate generated when all three processors are non-faulty which would otherwise stay forever in RMP.

Consider the modified receiver algorithm in Figure 40. When a message is received by the *receiver* process, it's *counterstamp* is compared with the *receiver.counter[s,d]* value.

- If *counterstamp* = *receiver.counter[s,d]* the message is sent to the neighbours and successor (intermediate triad) or sent to the neighbours and placed in VMP (destination triad) and the counter incremented.
- If *counterstamp* < *receiver.counter[s,d]* the message is a duplicate and may be discarded.
- If *counterstamp* > *receiver.counter[s,d]* this cannot occur as it requires two faulty processors in the source triad.

This mechanism removes the duplicates generated by diffusion at intermediate and destination triads which would otherwise greatly increase inter-triad message traffic.

```
process receiver:
    var m:message
    var counter[][]:sparse array of int:=0
    cycle
        if not authentic(m) then discard(m)
        else
            if m.counterstamp=counter[m.src][m.dest] then
                send_to_neighbours(m)
                if m.dest = me then deposit(VMP, m)
                else
                    send_to_successor(m)
                endif
                inc counter[m.src][m.dest]
            elseif m.counterstamp<counter[m.src][m.dest] then discard(m)
            elseif m.counterstamp>counter[m.src][m.dest] then
                serious_error(m)
                discard(m)
            endif
        endif
    end
end
```

Figure 40 Modified Receiver Algorithm

4.6 A Dual Processor Derivative

Consider again the system of Figure 23. In the replicated version, shown in Figure 41, each process c_i ($1 \leq i \leq 5$) is now replaced by a process pair C_i, such that $C_i \equiv \{C_{i1}, C_{i2}\}$ and each processor p_j ($1 \leq j \leq 4$) is replaced by a processor pair P_j such that $P_j \equiv \{P_{j1}, P_{j2}\}$ with C_{i1} mapped onto P_{j1} and C_{i2} onto P_{j2}.

As will now be shown, the failure of a processor in the source node results in no authentic double-signed messages being produced. The source node, when combined with the authentication mechanisms of the destination node, thus has *fail-silent* semantics. A dual-processor node may therefore used to implement a *fail-silent* processor such as that required by the Delta-4 XPA system discussed in Section 3.11.

Processors may be connected as shown in Figure 42. Assume that a process pair C_m is mapped onto processor pair P_1, and a process pair C_n is mapped onto processor pair P_4 and that C_m wishes to send a message to C_n. If we implement the algorithms of Figure 39 and Figure 40 in the manner of Figure 35, then the system will behave as follows.

In the absence of any faults, the *voter.majority* process will produce double-signed messages which will be received at the destination processor uncorrupted and passed to the destination process C_n.

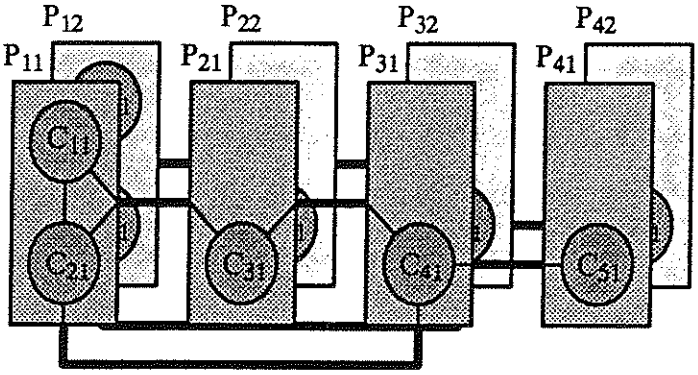

Figure 41 Replicated System

Thus in the absence of faults, the system will function correctly. Now consider system behaviour in the presence of faults:

If an external link fails (eg between P_{12} and P_{22}), one double-signed message will be lost. However, an equivalent double signed message will be sent on the external link between the corresponding neighbours (eg between P_{11} and P_{21}). This message will eventually be received at the destination processor uncorrupted and passed to the destination process C_n.

If an internal link fails in an intermediate pair (eg between P_{21} and P_{22}), each member will send-on the double-signed message it receives from it's predecessor (P_{11} and P_{12} respectively).

If a processor fails in an intermediate pair (eg P_{21}), then a double-signed message will be sent-on by it's neighbour (eg P_{22}).

Thus there are a class of failures which the system will mask; that is, any combination of external link failures, intermediate processor failures and intermediate internal link failures after which there remains a non-faulty path between the source and destination pairs. This is similar to the fault model represented by the connectivity properties *C1* and *C2* of Section 4.3. Such failures cannot be detected using the algorithms of Figure 39 and Figure 40 without modification.

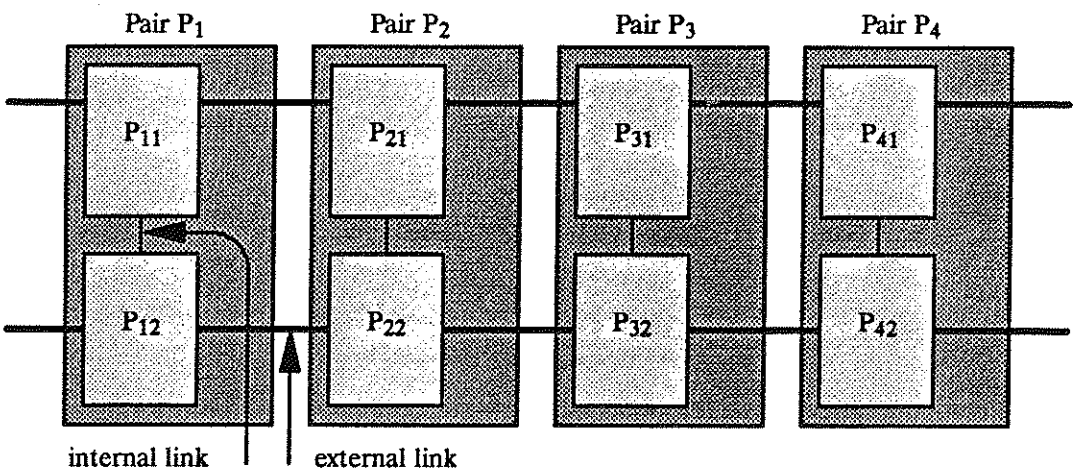

Figure 42 Processor Interconnections

Now consider the following failures:

Failure of a processor or internal link at the source pair may delay or prevent the *voter.majority* process finding a given message pair and hence delay or prevent the generation of the corresponding double-signed message.

Failure of a processor or internal link at the destination pair could prevent the correct functioning of the order process (see Chapter 5). This could result in a message being delayed or blocked in VMP (see Figure 35) and hence delay or prevent delivery of that message to the destination process C_n.

A combination of external link failures, intermediate processor failures and intermediate internal link failures which result in the source and destination pairs being no longer connected by a non–faulty path could delay or prevent delivery of a double–signed message to the destination processor.

If the message was transmitted asynchronously by the source process C_m, the fact that it was delayed or lost would be invisible to the non–faulty processor(s) of the source pair which would continue to function. Any subsequent attempt by C_m to send another message to C_n which resulted in the successful transmission of that message would be detected as an out–of–sequence message by the *receiver* process (*serious_error* in Figure 40) and discarded. All subsequent messages will also be discarded, so C_m would have *fail–silent* semantics. If the message was transmitted synchronously by the source process C_m (using some higher level protocol), C_m would block and therefore have *fail–silent* semantics.

Thus there are a class of failures which result in *fail–silent* behaviour of the source process.

4.7 Summary

This Chapter began by introducing a replicated processing model. It then presented a number of alternative physical architectures to support this model and showed how these architectures must be considered in parallel with the choice of voting algorithms. The subject of *public key cryptosystems* was then covered in preparation for a full description of the voting algorithms. The subject of duplicate removal was discussed in relation to the production of duplicates by faulty processors and the voting mechanism and a set of algorithms given for its implementation. Finally, a degenerate, *processor–pair* architecture was proposed to support *fail–silent* behaviour using the same mechanisms and algorithms. Note that none of the algorithms presented in this chapter require any form of clock synchronization.

Chapter 5 : The Order Process

5.1 Introduction

This chapter introduces the *order process* mechanism which is used in later chapters to prevent *state divergence*. Section 5.2 describes it's structure and relates it to the *Byzantine Generals Problem* of Section 2.4.2. Section 5.3 describes a set of *atomic broadcast* protocols which are used to implement the order process mechanism with varying degrees of fault-tolerance. Finally, Section 5.4 summarizes the contributions of this chapter.

Each processor P_{ij} has an order process O_{ij}, as shown in Figure 43 for the destination triad P_4 of Section 4.5. The function of each order process O_{4j} is to take a message from

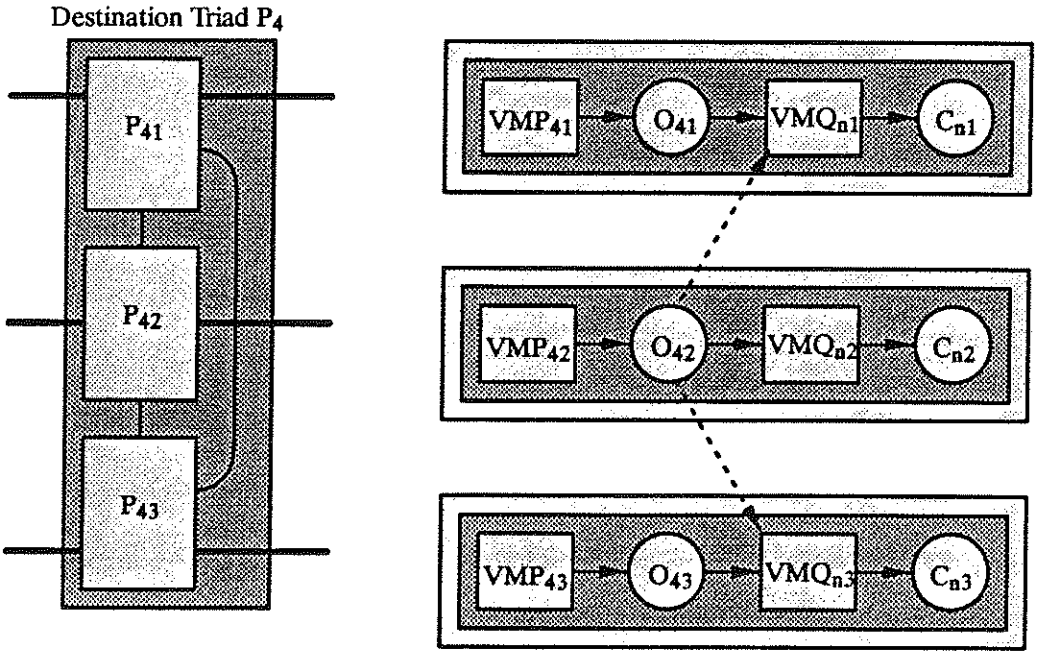

Figure 43 The Order Processes of a Triad

VMP_{4j} and broadcast it to all three of the VMQ_{nj}, as indicated for O_{42} by the dashed lines in Figure 43.

The order process thus faces a problem similar to that encountered in the *Byzantine Generals Problem* discussed in Section 2.4.2, namely the sending of an order by the commander (the order process) to his lieutenants (the VMQs). This type of broadcast is termed an *atomic broadcast* [64] and if implemented using a solution to the *Byzantine Generals Problem*, satisfies the *interactive consistency* conditions ***IC1*** and ***IC2*** of Section 2.4.2. This ability to satisfy ***IC1*** and ***IC2*** is expressed by the *atomic broadcast* properties of ***termination*** and ***atomicity***. In addition, the *atomic broadcast* possesses a third property of ***order***. This may be expressed in terms of *Byzantine Generals* as the lieutenants accepting their orders from three different commanders (the order processes) such that the sequence in which the orders are obeyed is consistent among the three lieutenants (the VMQs).

Although it may seem that with one commander (order process) and three lieutenants (VMQs) there will be a solution to this problem using oral messages ($n \geq 3m + 1$), one of the lieutenants and the commander are in fact the same person (the same processor). Thus a traitorous commander would also imply a traitorous lieutenant and with $n = 4$ there is no solution for $m = 2$ ($n < 3m + 1$) using oral messages. Any solution must therefore employ signatures and authentication ($n \geq m + 2$).

If the order processes use an *atomic broadcast* protocol to *send* the messages to the VMQs then the ***order*** property will ensure that the order of messages within the VMQs remains consistent.

In the absence of faults, each message will be broadcast three times, once by each O_{4j}, while a faulty order process could attempt to broadcast any number of duplicates. In addition, a faulty order process could attempt to broadcast an arbitrary, corrupted message. Corruption may be detected by authentication of the double-signature appended to the message by the source triad (see Section 4.5.2) while duplicates may be detected by comparison of the message *counterstamp* (protected by the double-signature) with a local *counter* (see Section 4.5.4). As these two techniques

have already been discussed in relation to the voting mechanism, they have been omitted from the following algorithms for the sake of clarity. The signature, authentication and duplicate removal mechanisms referred to in Section 5.3.3 are independent of, and additional to those employed by the voting mechanism.

5.2 Order Process Structure

In the following algorithms, the *order process* is composed of three sub–processes; *start* (S_{ij}), *relay* (R_{ij}) and *end* (E_{ij}) as shown in Figure 44.

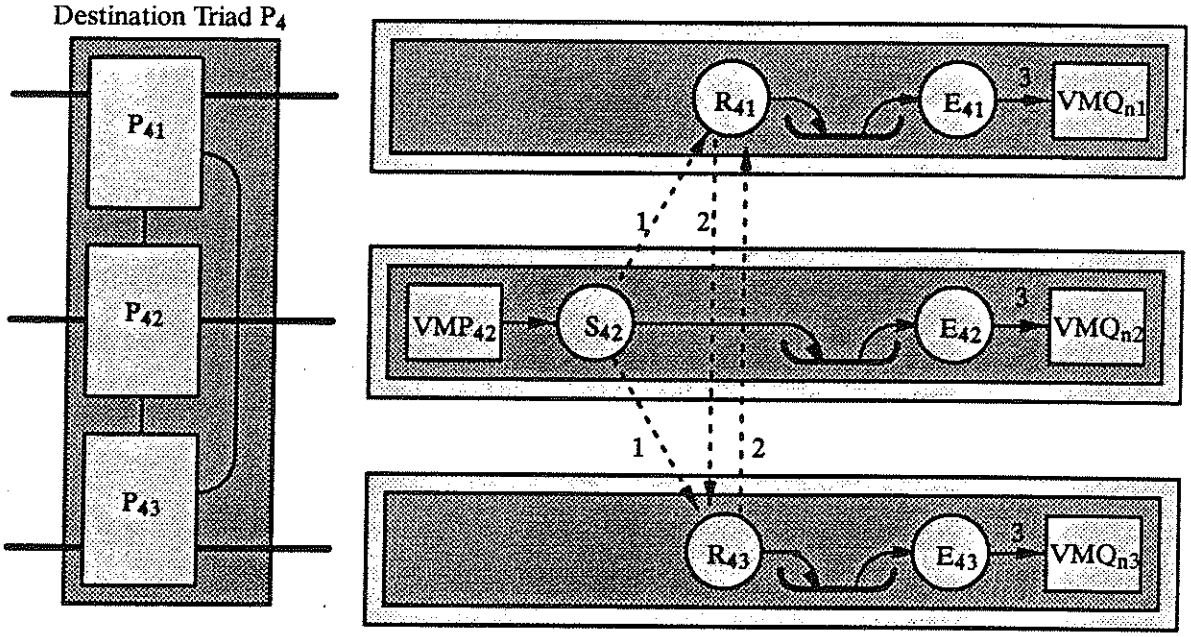

Figure 44 Structure of an Order Process

An atomic broadcast is performed in three stages:

stage 1 Each S_{ij} works independently, selecting a message from VMP_{ij}, placing it in a local pool then sending a copy to it's neighbours (only the path of a message from VMP_{42} is shown in Figure 44 for clarity) .

stage 2 Each R_{ij} receives this message and either rejects it or places a copy in a local pool and sends it to the other R_{ij}. The other R_{ij} would then still receive the message in the case where S_{ij} was faulty and had not send it directly.

stage 3 After some specified time interval, each E_{ij} removes the message from it's local pool and delivers it to the VMQ_{nj} of the destination process.

Thus the message will be placed in the local pool at either stage 1 (S_{ij}) or stage 2 (R_{ij}) and removed at stage 3 (E_{ij}).

Stage 1 and stage 2 as outlined above correspond to stage 1 and stage 2 of the *signed message* solution to the *Byzantine Generals Problem* described in Section 2.4.4. The additional stage 3 is necessary to satisfy the ***order*** property particular to the *atomic broadcast* abstraction as will be explained in the following sections.

All the following protocols conform to the model shown in Figure 44, they differ only in their criteria for rejecting a message (authenticity, timeliness etc) and in the length of the time interval. Although only the third protocol is suitable for the Byzantine world already assumed by the voting mechanisms, the first and second protocol are also presented, as they tolerate an ever increasing subset of Byzantine faults and each serves to explain a facet of the third.

5.3 Atomic Broadcast Protocols

A protocol to implement the *atomic broadcast* abstraction exhibits the following properties:

termination It delivers every message broadcast by a correct sender to all correct receivers after some known time interval (Δ).

atomicity It ensures that every message whose broadcast is initiated by a sender is either delivered to all correct receivers or none of them.

order It guarantees that all messages delivered from all senders are delivered in the same order at all correct receivers.

When employed in a fault–tolerant system, the protocol must continue to exhibit those properties even in the presence of faults. The protocol must therefore possess a degree of fault–tolerance at least as great as that of the system itself. Faults may be divided into three classes:

omission causes the component to omit to send a message which should be sent, according to the specification.

timing causes the component to send a message either too early or too late, according to the specification.

Byzantine causes the component to behave in any way which differs from the specification.

A family of protocols will now be presented, each tolerating a different class of faults. In all cases, a number of assumptions are made about the underlying hardware.

- Processors are connected by a point–to–point network.
- Faults do not cause network partitioning.
- A message sent between two correct adjacent processors over a correct link is delivered within δ time units.
- The clocks of correct processors are synchronized to within ϵ time units (see [53][54][55][56][57][58][59]).

Therefore, if a message sent by a correct processor p at local time t_p, over h correct links is delivered to a correct processor q at local time t_q then :

$$-\epsilon \leq t_q - t_p \leq h\delta + \epsilon$$

It is further assumed that each message carries a *timestamp* which gives its local time of departure t_p and that no correct processor will issue two messages with the same *timestamp*.

In the TMR architecture of Figure 44, if the link between P_{42} and P_{43} is faulty (may be considered as a failure of either P_{42} or P_{43}), the message removed from VMP_{42} at t_{42} ($=$*timestamp*) will be placed in the pool in P_{43} (via P_{41}, so $h=2$) at local clock time t_{43} where:

$$-\epsilon \leq t_{43} - timestamp \leq 2\delta + \epsilon$$

The order property of an atomic broadcast may then be implemented by ensuring that:

order In every correct processor, messages are delivered (to the VMQs) in the order of their timestamps or where equal, their senders identifier (a deterministic *tie–breaker*).

Because of the variation in transmission message times, a message may not be delivered to VMQ immediately upon arrival in the pool. Instead, it may be delivered only when it is no longer possible for a subsequent correct message to arrive bearing an earlier timestamp.

5.3.1 *First Protocol* (omission)

A processor which suffers from omission faults will either:

omission produce a valid message at the correct time or will produce no message (ie never corrupted, never early or late).

A message may be delivered only when it is no longer possible for a subsequent correct message to arrive bearing an earlier timestamp. In a system of n processors in which π are faulty, a message may be delivered at local clock time $t_{deliver}$:

$$t_{deliver} = timestamp + \pi\delta + (n - \pi - 1)\delta + \epsilon$$

where,

$timestamp$ = local time at which message was transmitted

$\pi\delta$ = time from transmission to acceptance by first correct processor

$(n-\pi-1)\delta$ = time for that correct processor to send the message to all the other correct processors

ϵ = maximum clock synchronization error

This is the worst case, in which all processors are connected in a pipeline, the sender is faulty and is separated from the nearest correct processor by all of the other faulty processors.

In the TMR architecture of Figure 44, this corresponds to the situation where P_{42} and the link to P_{43} are faulty. The message will be removed from VMP_{42} at t_{42} ($= timestamp$) and sent to P_{43} via P_{41} ($n=3, \pi=1$) and may therefore be removed from the pool in P_{43} at local clock time $t_{43} = t_{deliver}$ where:

$$t_{deliver} = timestamp + 2\delta + \epsilon$$

The algorithms to implement the first protocol for TMR are shown in Figure 45.

A message is discarded by the *relay* process only if it is a duplicate (in_pool(message)). Since a message cannot be corrupted, the function in_pool(message) may assume that two messages are duplicates if they have matching source and destination and matching timestamp. The *end* process may have to deliver several messages as it may have been scheduled by both the start and relay tasks simultaneously.

```
process order
    var pool:message_pool
    process start
        var m:message
        cycle
            remove_from_VMP(m)                  / select a message
            deposit(pool, m)                    / deposit in pool for end process
            send_on_both_links(m)               / copy to neighbouring relay processes
            schedule(end, t_deliver)            / schedule the end process to remove
                                                / message from the pool at time t_deliver
        endcycle
    end

    process relay
        var m:message
        cycle
            receive_on_a_link(m)                / receive message from start process
                                                / (or relay process)
            if (not in(pool, m))                / if not a duplicate
                deposit(pool, m)                / deposit in pool for end process
                send_on_other link(m)           / copy to other neighbour
                                                / (if received from start process)
                schedule(end, t_deliver)
            endif
        endcycle
    end

    process end
        var m:message
        var min:message_id
                                                / while there are deliverable messages
        while (messages_ready_to_deliver(pool) > 0) do
            min = min_timestamp_message_ready(pool)
            remove(pool, min, m)                / deliver oldest deliverable message
            deliver_to_appropriate_VMQ(m)
        od
    end

end
```

Figure 45 The First Protocol

5.3.2 *Second Protocol* (timing)

A processor which suffers from timing faults will either:

timing produce a valid message at the correct time or will produce a valid message at the wrong time (ie either on time, too early or too late but always correct).

The first protocol cannot tolerate *timing* faults. If a message was to be delayed by a faulty processor, it could be delivered to some correct processors before $t_{deliver}$ and to others after $t_{deliver}$. In the latter case, another message could be delivered before it thereby violating the required ***order*** property.

In the TMR architecture of Figure 44, this corresponds to the situation where P_{42} and the link to P_{43} are faulty. The message removed from VMP_{42} may be delayed by S_{42} such that it is delivered to the pool in P_{41} at local clock time $t_{41} < t_{deliver}$ and to the pool in P_{43} at local clock time $t_{43} > t_{deliver}$. By then, P_{43} could have already delivered a message bearing a higher timestamp.

In the second protocol, the time interval during which a message is acceptable is made proportional to h the number of processors which have already accepted it (equal to the number of links over which the message has already passed). A message is acceptable at local clock time t iff:

$$t_{min} \leq t \leq t_{max}$$

where,

$$t_{min} = timestamp - h\epsilon$$

$$t_{max} = timestamp + h\epsilon + h\delta$$

The $h\epsilon$ component is necessary to prevent a delayed message being rejected by some correct processors and accepted by others due to the clock skew ϵ between them.

```
process order
    var pool:message_pool
    process start
        var m:message
        cycle
            remove_from_VMP(m)                    / select a message
            deposit(pool, m)                      / deposit in pool for end process
            message.hop_count = 1
            send_on_both_links(m)                 / copy to neighbouring relay processes
            schedule(end, t_deliver)              / schedule the end process to remove
                                                  / message from the pool at time t_deliver
        endcycle
    end

    process relay
        var m:message
        cycle
            receive_on_a_link(m)                  / receive message from start process
                                                  / (or relay process)
                                                  / if not a duplicate and timely
            if (not in(pool, m) and t_local < t_max and t_local > t_min)
                deposit(pool, m)                  / deposit in pool for end process
                increment(message.hop_count)
                send_on_other link(m)             / copy to other neighbour
                                                  / (if received from start process)
                schedule(end, t_deliver)
            endif
        endcycle
    end

    process end
        var m:message
        var min:message_id
                                                  / while there are deliverable messages
        while (messages_ready_to_deliver(pool) > 0) do
            min = min_timestamp_message_ready(pool)
            remove(pool, min, m)                  / deliver oldest deliverable message
            deliver_to_appropriate_VMQ(m)
        od
    end

end
```

Figure 46 The Second Protocol

In a system of n processors in which π are faulty, a message may now be delivered at local clock time

$$t_{deliver} = timestamp + \pi\delta + \pi\epsilon + (n - \pi - 1)\delta + \epsilon$$

where,

$\pi\delta + \pi\epsilon$ = time from transmission to acceptance by first correct processor

In the TMR architecture of Figure 44, if P_{42} and the link to P_{43} are faulty, the message is removed from VMP_{42} at t_{42} (= *timestamp*) and sent to P_{43} via P_{41} ($n=3$, $\pi=1$) and may therefore be removed from the pool in P_{43} at local clock time $t_{43} = t_{deliver}$ where:

$$t_{deliver} = timestamp + 2\delta + 2\epsilon$$

The algorithms to implement the second protocol for TMR are shown in Figure 46. They differ from those of the first protocol only in that the *start* task appends a *hop count* to the message and the *relay* task incorporates two additional acceptance tests ($t < t_{max}$ *and* $t > t_{min}$) and increments the *hop count*.

5.3.3 *Third Protocol (Byzantine)*

A processor which suffers from Byzantine faults will either:

Byzantine produce a valid message at the correct time or behave arbitrarily.

A processor affected by a Byzantine fault could defeat both the previous protocols, by modifying the *timestamp* for example. In the third protocol, signatures and authentication (see section 4.4) are employed to prevent this happening.

It is assumed that,

signature Each non-faulty processor possesses a unique signature which with high probability, is unforgeable by any other processor.

authentication Each non-faulty processor can authenticate a signature with high probability.

These assumptions are similar to assumption *A4* in the solution to the *Byzantine Generals Problem* using signed messages. Three procedures are assumed.

sign(m) append the local processor's signature to the message m

co-sign(m) append the local processor's signature to the list of signatures on m

authenticate(m) verify the authenticity of the message m

The message does not carry a hop count as this may be determined by counting the number of appended signatures.

If the time to authenticate is ignored, t_{min}, t_{max} and $t_{deliver}$ are the same as for the second protocol.

The algorithms to implement the third protocol for TMR are shown in Figure 47. The function authenticate(message) detects a corrupted message and discards it, so the function in_pool(message) may still assume that two messages are duplicates if they have matching source and destination and matching timestamp. However, a faulty processor could send two different authentic messages with the same timestamp which could arrive at the two non-faulty neighbours in a different order. Without any additional mechanisms, the second message would be regarded by in_pool(message) as a duplicate and discarded, resulting in inconsistent VMQs. The different_to_pool(message) function detects this case and marks the message as bad so that the *end* process will not deliver it to the VMQ.

```
process order
    var pool:message_pool
    process start
        var m:message
        cycle
            remove_from_VMP(m)                    / select a message
            deposit(pool, m)                      / deposit in pool for end process
            sign(m)                               / sign message and
            send_on_both_links(m)                 / copy to neighbouring relay processes
            schedule(end, t_deliver)              / schedule the end process to remove
                                                  / message from the pool at time t_deliver
        endcycle
    end
process relay
        var m:message
        cycle
            receive_on_a_link(m)                  / receive message from start process
                                                  / (or relay process)
                                                  / if uncorrupted and timely
            if ((authenticate(m) and t_local < t_max and t_local > t_min) then
                if (not in(pool, m)) then         / if not a duplicate
                    deposit(pool, m)              / deposit in pool for end process
                    co-sign(m)                    / co-sign message and
                    send_on_other link(m)         / copy to other neighbour
                    schedule(end, t_deliver)      / (if received from start process)
                else                              / If a different message exists with the
                    if different_to(pool, m) then / same source, destination and
                        mark_as_bad(pool, m)      / timestamp, cause both to be
                                                  / discarded
                    endif
                endif
            endif
        endcycle
    end
    process end
        var m:message
        var min:message_id                        / while there are deliverable messages
        while (messages_ready_to_deliver(pool) > 0) do
            min = min_timestamp_message_ready(pool)
            remove(pool, min, m)                  / deliver oldest deliverable message
            if not (marked_as_bad(pool, m)) then  / (if sender is not faulty)
                deliver_to_appropriate_VMQ(m)
        od
    end
end
```

Figure 47 The Third Protocol

5.3.4 *Performance*

In the absence of faults, the number of messages sent per broadcast per link is

$$2 - \frac{|\mathcal{P}| - 1}{|\mathcal{L}|}$$

where,

$|\mathcal{P}|$ = The number of processors

$|\mathcal{L}|$ = The number of links

For sparsely connected networks this is close to 1, while for highly connected ones it is close to 2.

When the third algorithm is applied to the TMR node architecture of Chapter 4, the termination time becomes

$$(n - 1)\delta + (\pi + 1)\epsilon = 2\delta + 2\epsilon$$

5.4 Summary

The *order process* was introduced as a mechanism for ensuring that the order of messages within the message queues (VMQs) of process replicas remains consistent. Then the *atomic broadcast* abstraction was discussed in relation to the *Byzantine Generals Problem* of Section 2.4.2 and a number of *atomic broadcast* algorithms given to implement the *order process* with varying degrees of fault-tolerance. Finally, the performance of the algorithms when applied to the system of Chapter 4 was investigated.

Note that because the atomic broadcast takes place within the triad, clock synchronization is required between the processors within a triad but is not necessary between triads, ie there is no need for global clock synchronization.

In the absence of faults, three atomic broadcasts of the same message will take place, so there is a requirement for the removal of duplicates. This could be accomplished by processing the output of the *order process* using the sequence number contained within the original double-signed message as described in Section 4.5.4. However, it should be possible to intercept duplicates earlier by modifying the *start* and *relay* processes (Figure 45, Figure 46 and Figure 47) to include an acceptance test based on the sequence number. This would then obviate the need to compare *timestamps* in the relay process.

Note that the order process satisfies only ***O1*** of Section 2.2. To satisfy ***O2***, additional mechanisms are required to implement total ordering (see Section 2.5). If the message sequence number is used to implement a logical clock [5] then the algorithms for duplicate removal must be modified (see Section 4.5.4) for the case where *counterstamp* > *voter.receiver.counter[s,d]*.

Chapter 6 : Non-determinism in the Enhanced Model

6.1 Introduction

In this section, the restrictive state machine model introduced in Section 4.2 is expanded to include enhanced message selection, asynchronous external events, non–deterministic processing and semaphores. State divergence is prevented in all cases by exploiting the ***order*** property of the *order process* (see Section 5.1). In Section 6.7, the overheads associated with this technique are examined.

6.2 Enhanced Message Selection

In the model presented in Section 4.2, a process may select any input message for processing. As a consequence, it is acceptable in the replicated version for a process to accept messages in an order determined by the order process. However, it may be desirable for a process to exercise some control over the selection of messages. This section examines a range of possible selection criteria in terms of classes of input and shows how the order process alone cannot handle such inputs.

6.2.1 *Blocking Input*

The process may specify a subset of clients from which it is prepared to accept a message at any given point, but must accept one before continuing. This could happen when the process is acting as a resource allocator server for example, and under some condition is prepared to listen to only some subset of its clients. If the message supplied first by the order process is not in the subset then we have deadlock. Two forms of blocking input are considered; *selective* and *alternative*.

(i) Selective Input

The client C_i from which a message is to be accepted is specified by the process S_k, as shown in the algorithm of Figure 48.

```
process Sk:
    var m:message
...
    receivefrom(Ci, m)
    action(m)
...
end
```

Figure 48 Selective Input

(ii) Alternative Input

The set of clients $\mathcal{S} = [C_i...C_j]$ from which a message may be accepted is specified by the process S_k, as shown in Figure 49. The value returned by a call to the function

```
process Sk:
    var m:message,
        S:setof client
...
    S := [Ci...Cj]
    switch (receivefrom(S, m))
        case Ci:
            actioni(m)
        ...
        case Cj:
            actionj(m)
    end
...
end
```

Figure 49 Alternative Input

receivefrom($\mathcal{S}$, m) identifies the sender of the selected message and is used to perform an appropriate action.

6.2.2 *Non–Blocking Input*

The process may specify a range of clients from which it is prepared to accept a message but may continue execution if no message is available. When such a process is replicated, either all the replicas must accept the same message or all of the replicas must continue execution. Since messages can arrive at replicas at different times, just preserving order is not sufficient to cope with non–blocking input. Three variants of non–blocking input are considered.

(i) Conditional Input

The client C_i from which a message is to be accepted is specified by S_k, as in the algorithm of Figure 50. Whereas in Figure 48 the receivefrom(C_i, m) function is assumed

```
process Sk:
    var m:message
...
    if receivenowaitfrom(Ci, m)
    then
        action(m)
    else
        altaction()
    endif
...
end
```

Figure 50 Conditional Input

to block until an appropriate message is received, here a Boolean value is returned by receive*nowait*from(C_i, m); **true** if the message is present, **false** if it is not. Thus in the absence of a message from C_i, the process continues execution with altaction().

(ii) Timed Input

With timed input, the alternative action is performed only if a message does not arrive before the expiry of a timeout as shown by Figure 51. So a Boolean value is returned by receivefrom(C_i, m, t); **true** if the message is present, **false** if it is not.

```
process Sk:
    var m:message,
        t:timeval
...
    t := <period>
    if receivefrom(Ci, m, t)
    then
        action(m)
    else
        altaction()
    endif
...
end
```

Figure 51 Timed Input

(iii) Timed Alternative Input

The set of clients $\mathcal{S} = [C_i...C_j]$ from which a message may be accepted is specified by the process S_k, as shown by Figure 52. But if an input does not arrive before the expiry of a

```
process Sk:
    var m:message,
        t:timeval,
        S:setof client
...
    S := [Ci...Cj]
    t := <period>
    switch (receivefrom(S, m ,t))
        case Ci:
            actioni(m)
        ...
        case Cj:
            actionj(m)
        case null
            altaction()
    end
...
end
```

Figure 52 Timed Alternative Input

timeout, some alternative action is performed. The identity of the successful client (or **null** if timeout) is returned by the call to receivefrom($\mathcal{S}$, m, t).

6.2.3 *Replicating the Enhanced Model*

Two forms of enhancement are proposed; blocking and non–blocking input on specified clients. When replicating the enhanced model, the order process solution discussed in Chapter 5 is not sufficient to deal with such inputs; it could cause deadlocks for the former case and state divergence for the latter. Deadlocks can be prevented if a process is allowed to perform an ordered search on the message queue for the first acceptable message. Since messages in the queues of replicas are ordered identically, this solution will work. Thus blocking inputs (Figure 48 and Figure 49) can be handled easily. More complex mechanisms are required to solve the second case. In the following sections, a general mechanism will be presented which resolves non–determinism for both cases.

6.2.4 *The Generic Input Function*

Both unspecified and selective input are special cases of alternative input ($\mathcal{S} = \mathcal{U}$, where $\mathcal{U}$ is the set of all possible clients and $\mathcal{S} = C_i$ respectively). Conditional input is a special case of timed input ($t = 0$). Alternative ($t = \infty$) and timed ($\mathcal{S} = C_i$) input, and hence unspecified, selective and conditional input are all special cases of timed alternative input (See Figure 52).

Since all the input forms discussed so far may be represented by timed alternative input, resolving non–determinism for the generic input function receivefrom($\mathcal{S}$, m ,t) will also resolve non–determinism for all other cases.

Consider the algorithm of Figure 53 which describes how the generic input function can be implemented using blocking input.

The generic input function waits for a maximum duration t to receive a message. Assuming timeout does not occur, the blocking function RECEIVEFROM($\mathcal{S}$, m) returns the first message in the queue for which $C_i \in \mathcal{S}$. Since the order of queues is consistent

among replicas (ensured by the order process), so is the choice of message. The state and subsequent behaviour of the server are therefore also consistent among replicas.

```
procedure receivefrom
    (𝒮:setof client, m:message, t:timeval):
    var c:client
begin
    within t do
        {c := RECEIVEFROM(𝒮, m)}
    timeout
        {sendto(self, timeout)
        c := RECEIVEFROM([𝒮, self], m)}
    od
    if (c = self)
    then
        return null
    else
        return c
    endif
end
```

Figure 53 The Generic Input Function

If no message is received within the duration t, then the timeout clause will be executed. The function then attempts to send a *timeout* message to itself. If a *majority*

Unspecified Input	$\mathcal{S} = \mathcal{U}$	$t = \infty$
Selective Input	$\mathcal{S} = C_i$	$t = \infty$
Alternative Input	$\mathcal{S} = C_i...C_j$	$t = \infty$
Conditional Input	$\mathcal{S} = C_i$	$t = 0$
Timed Input	$\mathcal{S} = C_i$	$t = \tau$
Timed Alternative Input	$\mathcal{S} = C_i...C_j$	$t = \tau$

where τ = timeout value

Figure 54 Input Mapping

of replicas time-out (see Section 4.5.2), then a *timeout* message will be majority voted and will arrive at all the replicas. The order processes ensure that messages are identically ordered in all the replica queues. So RECEIVEFROM([𝒮, self], m) will return the same message to each replica; either a message from a member of $\mathcal{S}$ or the *timeout* message. The problem is therefore transformed to that of alternative input from

$\mathcal{S} = [C_i...C_j, self]$ for which consistency is maintained. The complete mapping of input types onto the generic input function receivefrom($\mathcal{S}$, m, t) is shown by Figure 54.

6.2.5 *Prioritized Input*

In Section 6.2.4, it was stated that the message returned by a call to RECEIVEFROM($\mathcal{S}$, m) is the first in the queue for which $C_i \in \mathcal{S}$; all messages assume equal priority. However, it may be desirable to impart some other ordering strategy based on a concept of urgency, for example, where the level of priority is derived from the client identifier or perhaps from the message itself. This form of input may be called prioritized alternative input. The replica must then search the queue for the highest priority message for which $C_i \in \mathcal{S}$. Although the order of the queues is guaranteed to be consistent among replicas, the time at which the queues are updated relative to the call to RECEIVEFROM($\mathcal{S}$, m) is non-deterministic. So the set of messages present in the queue at the time of the search need not be identical among replicas. A call to RECEIVEFROM($\mathcal{S}$, m) could therefore return a different client identifier in each replica, causing state-divergence.

The problem is similar to that of conditional input where the presence or absence of a message may affect the future behaviour of a process. In fact, prioritized alternative input may be expressed as a nested sequence of conditional inputs. The problem may therefore be solved by a nested sequence of calls to receivefrom($\mathcal{S}$, m ,t), with t=0. However, this solution would place a large load on the communication medium due to the continuous production of *timeout* messages. A more efficient implementation of RECEIVEFROM($\mathcal{S}$, m) which can handle prioritized messages is now presented.

The problem may be solved by using a self-directed message to mark the queue, as shown in Figure 55. The set of messages which arrive before the *marker* is consistent among replicas. So an ordered search which is restricted to this set will always produce a consistent result.

A call is made to RECEIVEREADYFROM($\mathcal{S}$) which blocks until a message from $C_{i \in \mathcal{S}}$ arrives in the queue then returns without removing it. The function then sends a *marker* message to itself. When a *majority* of *marker* messages is achieved, a copy will be placed in the queue by the order process. A further call is made to RECEIVEREADYFROM(self) which blocks until the *marker* message arrives in the queue then returns without removing it. At this stage, each replica queue contains at least one message from $C_{i \in \mathcal{S}}$ followed by the *marker* message and the replica queues are identical up to and including the marker message. An ordered search of the restricted queue is then performed to identify the highest priority message. Finally the *marker* and highest priority message are both removed from the queue and the latter returned to the caller.

```
procedure RECEIVEFROM
    (S:setof client, m:message):
    var c:client,
begin
    RECEIVEREADYFROM(S)
    sendto(self, marker)
    RECEIVEREADYFROM([self])
    c :=    <Identity of highest priority
            message in queue ahead of
            the mark>
    receivefrom(self, m)
    receivefrom(c, m)
    return c
end
```

Figure 55 Prioritized Alternative Input

6.3 Asynchronous External Events

Practical applications often require the use of asynchronous signals (eg an alarm signal) to force a process to immediately stop its current activity and perform some alternative action. However, if the process replicas were stopped independently, they could be stopped at different points in their execution, resulting in a difference in their

internal states. If the subsequent behaviour of the alternative action is dependent on this state then divergence could occur.

The same problem was encountered in Section 6.2.2 in relation to non-blocking input and was solved by the *generic input function* using a self-directed message to order timeout with respect to other messages as described in Section 6.2.4. If we transform an external event into a message from a system client e, it too may be ordered with respect to other messages. Then by adding e to the set of acceptable clients, whenever the *generic input function* is called, we can ensure that all replicas receive the event message (and hence respond to the event) at identical points in their execution. We may adjust the priority of an event in relation to other events and messages using the method discussed in Section 6.2.5, but this requires the transmission of a self-directed message to establish priority. If the original client set expressed no priority this becomes an additional overhead.

```
procedure pre-emption_point(event_handler:procedure):
begin
    var m:message
    if receivenowaitfrom(e, m)
    then
        event_handler(m)
    endif
end
```

Figure 56 Conditional Input

Although this solution will work, the response time to an external event is dependent on the granularity of input statements. If the opportunities for message input are infrequent, it may be desirable to add additional *pre-emption* points at which a signal may be accepted by explicitly performing conditional input from e, as shown in Figure 56. This is equivalent to calling the *generic input function* with $s = e$ and $t = 0$ where a timeout means *no event detected*. So every time a pre-emption point is executed when an event has not occurred, the process is delayed waiting for the self-directed message to arrive, which makes this type of *pre-emption* point a

considerable overhead. This technique is similar to that used in the *leader–follower* model of Delta–4 XPA [50][51][52] described in Section 3.11 which does not suffer the same overhead due to its restricted failure assumptions.

We can eliminate the overhead by taking advantage of the periodic nature of pre-emption points to perform the transmission of the self–directed message in parallel with the normal processing of the replica.

```
procedure pre-emption_point(event_handler:procedure):
begin
    var m:message
    if RECEIVEFROM([c, self], m) = = c
    then
        event_handler(m)
        sendto(self, null)
    endif
end
```

Figure 57 Pre-emption Point

A self–directed message is sent at the end of every pre-emption point, as shown in Figure 57. If no event occurs before that message is entered into the queue then the next pre-emption point will receive that message and execution will continue. Only one such self–directed message will be *in transit* at any point in time. After an event occurs, an event message will be produced and entered into the queue. This event message will be ordered with respect to the self–directed message. Therefore either all pre-emption point replicas will receive the event message and respond to the event or all will receive the self–directed message. In the latter case, the event message will be received by the next pre-emption point. Pre–emption points implemented in this way handle non–deterministic processing more efficiently than the method of adding c to the set of acceptable clients whenever the *generic input function* is called, as described earlier in this section.

The price we pay for eliminating the overhead is the introduction of an additional latency into the response time to an external event. The RECEIVEFROM(s,m) procedure

referred to in Section 6.2.4 (ie no priority) is used in preference to that of Figure 55 to avoid the transmission of yet another self-directed message which would re-introduce the overhead we seek to eliminate. Note that a self-directed message must be sent during initialization to prevent the first pre-emption point blocking on the call to RECEIVEFROM([c, self], m) until the occurrence of an event.

6.4 Non-deterministic Processing

Consider the non-deterministic program segment shown in Figure 58.

```
t := timer + timeval
while timer < t do
    i := i + 1
    send(i)
od
```

Figure 58 Program Segment

As it stands, such a program cannot be replicated without causing divergence of state as the value of *i* at timeout need not be identical among replicas. The timeout may be thought of as an external event and handled by one of the methods described in Section 6.3. The pre-emption point method is preferable as it is more efficient and there may be no input statements within the scope of the timed construct.

```
t := timer + timeval
schedule(alarm(t))
forever do
    pre-emption_point(exit)
    i := i + 1
    send(i)
od
```

Figure 59 Pre-emption Point

The action to be taken in response to a timeout message is to *exit* the timed construct as shown in Figure 59 which implements the timed construct of Figure 58.

6.5 Semaphores

It was stated in Chapter 3 that processes communicate only via message–passing. However, some applications could be implemented more efficiently if processes sharing a processor and having a common address space were also able to share memory. Semaphores would typically be the mechanism used to control access to this shared memory.

An implementation of semaphores must contain indivisible *request* and *release* operations. In a non–replicated system they could be part of the operating system, in the form of non–interruptable primitives or they could be provided by the underlying hardware component. When such a system is replicated, the order in which *requests* are *granted* could be different in each replica and could thereafter cause a divergence of state. It is therefore necessary to ensure that semaphore operations are indivisible across all replicas.

A call to a semaphore primitive may be viewed as a request to a server (the operating system) to perform some operation (lock/unlock) on a shared resource. If such a call were translated into a message to a *real* server, then requests would be ordered by the order process of each replica. The server would conform to the basic model of Section 4.2, so the order in which *requests* were *granted* would be consistent among replicas. The *grant* message would then be translated to form the reply to the original call.

6.6 Real-Time Clocks

Application programs may require access to a real-time clock. In a uni-processor system, the application would typically call an operating system primitive which would simply return the current value of the processor clock. In a distributed system, two processes on distinct processors which read their own clocks at approximately the same point in (real) time should obtain *approximately* the same reading. This is usually accomplished by some form of clock synchronization [53][54][55][56][57][58][59]. In a replicated system however, all process replicas on non-faulty processors which read their own clocks at the same point in their execution must obtain exactly the same reading. Even if the clocks could be synchronized exactly, the replicas read their clocks independently and hence could obtain inconsistent readings.

A typical clock synchronization may be performed in three stages.

stage 1 Processors cooperate to execute an interactive consistency algorithm to ensure that all non-faulty processors have a consistent view of the current clock values C_{i0}, of all other non-faulty processors.

stage 2 Each processor independently performs a deterministic calculation on this set of values to obtain an a single agreed value C_{new}.

stage 3 Each processor adds an offset to its own clock to take account of the difference between the agreed value calculated in stage 2 and its own value, as disseminated in stage 1 ($C_i = C_i + (C_{new} - C_{io})$).

Thus the clocks of all non-faulty processors which participate in stage 1 are synchronized to within some known bound ϵ.

One possible solution to the problem of ensuring consistent clock readings among replicas is to repeat stages 1 and 2, and use the single agreed value. However, step 3 of the underlying clock synchronization requires the addition of an offset to the current processor clock. Thus the synchronized clock will appear to run a little faster than

real–time. As the clock synchronization period is reduced, to reduce ϵ, this speed–up becomes more pronounced. If the application requires a *true* real–time clock, this method may be inappropriate.

Apparent clock speed–up is a direct result of clock synchronization. So if stages 1 and 2 were to use the unsynchronized clocks, consistent clock readings would still be obtained but would more accurately reflect the passage of real time.

This may be accomplished by not modifying the clock in stage 3′ of the clock synchronization algorithm, but instead merely calculating the offset. This offset would then be added to the current clock value before dissemination in stage 1′ of the clock synchronization algorithm but would be ignored during stage 1 of ensuring consistent reading of the real–time clock.

Clock synchronization becomes:

stage 1′ Processors cooperate to execute an interactive consistency algorithm to ensure that all non–faulty processors have a consistent view of the current clock values ($C_{i0} + C_{ioff}$), of all other non–faulty processors.

stage 2′ Each processor independently performs a deterministic calculation on this set of values to obtain an a single agreed value C_{new}.

stage 3′ Each processor calculates an offset to its own clock to take account of the difference between the agreed value calculated in stage 2 and its own value, as disseminated in stage 1 ($C_{ioff} = C_{new} - (C_{i0} + C_{ioff})$).

Ensuring a consistent real–time clock remains:

stage 1 Processors cooperate to execute an interactive consistency algorithm to ensure that all non–faulty processors have a consistent view of the current clock values C_{i0}, of all other non–faulty processors.

stage 2 Each processor independently performs a deterministic calculation on this set of values to obtain an a single agreed value C_{new}.

The clock may be viewed as a server providing a clock service. A call to an operating system primitive to read the current value of the clock could cause a message to be sent to the clock server which would ensure consistent clock readings by using the steps outlined above. The message returned by the server would then be transformed into the primitive return value. The clock server would also perform the clock synchronization algorithm.

Having structured the real-time clock as a server, another method for ensuring consistency may be used. The method is similar to the solution for timeouts (see Section 6.2.4) and asynchronous events (see Section 6.3). The principle of this method is to implement a *logical real-time clock* and to order clock *read* requests with respect to ticks of that clock.

Each server replica periodically and independently attempts to send a *tick* message to itself. Timing is with respect to its own independent, unsynchronized clock. When a *majority* of replicas attempt to *tick* (see Section 4.5.2), a *tick* message will be majority voted and will arrive at all the replicas. The clock server then *ticks* by incrementing a clock variable. Note that the value of the clock variable will be identical to the *tick* message sequence number (see Section 4.5.4).

Clock *read* requests arrive at the server as messages. The order processes ensure that messages are identically ordered in all the replica queues. So *read* messages will be consistently ordered with respect to *tick* messages, and replicas will read consistent clock values.

6.7 Performance

The *generic input function* presented in Section 6.2.4 relies on the order process, which is implemented using an atomic broadcast protocol such as that presented in Section 5.3.3. Such a protocol has a termination time of:

$$\Delta_t = 2\delta + 2\epsilon$$

where δ is the maximum time required to transmit a message between two adjacent processors in a triad and ϵ is the maximum possible synchronization error between the clocks of those processors.

Thus every message transmitted from some process P_a to some other process P_b will incur an additional latency of Δ_t due to the actions of the order process of the processor executing P_b. However, the order process will atomically broadcast this message as soon as it arrives at the destination processor, whether or not P_b is ready to receive it. So if P_b does not become ready until at least Δ_t after the message arrives at the destination processor, it would not be affected by this additional latency.

If the generic input function detects a timeout while waiting for a message, it sends a *timeout* message to itself (see Section 6.2.4). This message is atomically broadcast by the order process and therefore a further Δ_t expires before an actual timeout can occur. If the intended message arrives before this timeout message, no timeout will occur. Thus the effective timeout period is:

$$t + \Delta_t$$

where t is the timeout value parameter given in the call to the generic input function.

If the actual timeout period must be t, then the value $t - \Delta_t$ must be passed as a parameter to the generic input function. It follows therefore that timeout periods of less than Δ_t are not possible.

When a process executes a prioritized input construct an additional self-directed *marker* message must be sent and received before the input can take place (see Section 6.2.5). Thus a process will be delayed by Δ_t when executing such a construct.

External events must be converted into messages by the system. This will require an atomic broadcast, resulting in a latency of Δ_t before the event message becomes *visible* to the process. If external events are handled using pre-emption points, implemented as in Figure 56, there will be an overhead of Δ_t for each pre-emption point. If the time between pre-emption points is Δ_p, there will be an event latency of between Δ_t and $\Delta_t + \Delta_p$. That is an average of:

$$\Delta_t + \frac{\Delta_p}{2}$$

If however, external events are handled using pre-emption points, implemented as in Figure 57, there will be no significant overhead but the latency will increase to between Δ_p and $2\Delta_p$. That is an average of:

$$\frac{3\Delta_p}{2}$$

Note that that the time between pre-emption points, Δ_p cannot be less than Δ_t, the time taken to transmit the self-directed message.

When a process executes a non-deterministic construct such as that shown in Figure 58 of Section 6.4, a pre-emption point must be executed every iteration, as shown in Figure 59. Since the minimum time between pre-emption points, Δ_p cannot be less than Δ_t, the iteration rate of such a construct will be limited to $1/\Delta_t$.. The effective timeout period will be *t* plus an additional event (timeout) message latency, the value of which depends on which method is used to implement pre-emption points.

That is an average of:

$$\Delta_t + \frac{\Delta_p}{2} \qquad \textit{or} \qquad \frac{3\Delta_p}{2}$$

If the execution time of the construct (minus pre-emption point) is small then the iteration rate will be determined largely by the minimum time between pre-emption points, $\Delta_p \simeq \Delta_t$ and the event (timeout) message latency for both methods will tend towards:

$$\frac{3\Delta_t}{2}$$

If the actual timeout period must be t, then value passed to the alarm call in Figure 59 must be reduced accordingly. Note that the actual timeout period may occur over a range of values.

The semaphore server of Section 6.5 requires a *request* message before it will send a *grant* message. Both of these messages must be atomically broadcast, so the minimum time between the issue of a *request* and the receipt of *grant* is $2\Delta_t$. Similarly the time between the issue of a *release* message and receipt of a *grant* message by another requesting user is $2\Delta_t$. Thus there is an overhead of $2\Delta_t$ every time the ownership of a resource is transferred from one user to another using semaphores.

The mutual exclusion implemented by the server is achieved by atomically broadcasting *requests*. If the semaphore server is implemented on the same processor triad as a user, then a failure of the processor would imply failure of a user replica and a server replica. So the *grant* and *release* messages need not be atomically broadcast and the overhead is then reduced to Δ_t.

Real-time clocks may be implemented using a clock server. The server may achieve consistency by performing a clock synchronization type algorithm or by using *logical real-time clocks*. In the former case, a request to read the clock will be delayed by Δ_t for

the request message to be ordered, and by a further Δ_t for the agreement protocol, a total of $2\Delta_t$. In the latter case, a request to read the clock will be delayed only by Δ_t for the request message to be ordered.

6.8 Summary

This Chapter proposed several enhancements to the state machine model; enhanced message selection, asynchronous external events, non–deterministic processing, semaphores and real–time clocks. A single well–defined mechanism, the *order process*, was then used to prevent state divergence.

This technique may introduce a processing or latency overhead proportional to the atomic broadcast time, Δ_t (and/or the time between pre–emption points, Δ_p), as described in Section 6.7. This overhead will be invisible in many applications. However, if it is deemed to be excessive for a given application, then either the processing model (e.g. *state machine model*) or the failure assumptions (e.g *fail–silent* as in Delta–4 XPA) must be restricted.

Chapter 7 : Non-determinism in Programming Languages

7.1 Introduction

In this section we investigate the sources of non-determinism in real systems and show how they may be mapped onto the generic input function. Where the language dictates synchronized communication (eg OCCAM channels) we assume the existence of a handshake protocol which employs asynchronous communication at a lower level†. It is at this lower level that we consider the mappings.

7.2 OCCAM

Occam [66] is based on the process model of computing. A process is an independent computation with its own program and data. Processes can communicate only through message passing, there is no sharing of data. Messages are input from and output to named channels. An output statement succeeds only when a process executes an input statement on the same channel while an input statement succeeds only when a process executes an output statement on the same channel. Thus communication is synchronized.

The **ALT** construct, shown in Figure 60, is composed of a number of input statements, only one of which may succeed. Each input statement is followed by an action statement. An action statement is executed if and when its corresponding input statement succeeds. If more than one alternative is ready when the **ALT** construct is executed, selection between them is arbitrary. One of the input statements may be replaced by either a **SKIP** statement, which succeeds immediately if no other input can

† This must be true of any distributed implementation of such a language and is not peculiar to replicated systems.

succeed or, a watchdog timer input statement, which succeeds if no other input statement succeeds before completion of the specified period. The **SKIP** statement is therefore equivalent to a timer input statement with a zero time value.

```
CHAN chan_i, chan_j :
TIMER clock :
INT n, start:
SEQ
    clock ? start
    ALT
        chan_i ? n
            action_i(n)
        ...
        chan_j ? n
            action_j(n)
        clock ? AFTER start PLUS t
            altaction()
```

Figure 60 The ALT Construct

The **ALT** construct may be transformed into the generic input function receivefrom($\mathcal{S}$, m, t), by mapping each channel onto a member client of the set $\mathcal{S}$ and the watchdog input time value onto the time parameter t. The value returned by receivefrom($\mathcal{S}$, m, t) is mapped onto the selection of an action$_i$() or the altaction() (timeout).

Priority based on channels may be expressed by the **PRI ALT** construct. The **PRI ALT** construct may be transformed into the generic input function by additionally mapping the lexical order of each input statement onto the priority of the message received on the specified channel.

7.3 Ada

Concurrency in Ada [67] can be expressed using the client/server model of computing. An Ada task, which may act as both a client and a server, is an independent computation with its own program and data. A task advertises its services as a collection of entry procedures‡. Tasks communicate by calling each others entry procedures. A call to an entry procedure succeeds only when the recipient executes an accept statement for that entry procedure. An accept statement succeeds only when the

```
task server is
    entry service_i();
    ...
    entry service_j();
end server;

task body server is
begin
    select
        accept service_i() do
            action_i();
        end;
        ...
        or
        accept service_j() do
            action_j();
        end;
        or
        delay <period>;
            altaction();
    end select;
end server;
```

Figure 61 The Select Statement (accept)

specified entry procedure is called by another task. Thus the two tasks come together at what is called a rendezvous. The client must specify the server entry procedure.

‡ *Tasks may also communicate via local copies of shared variables but the possibility of a successful implementation of this facility in a distributed environment is questionable.*

However, while the server has control over which entry procedure is accepted, it cannot even identify let alone specify the client.

The **select** statement, shown in Figure 61, may be composed of a number of **accept** statements, only one of which may succeed. Each **accept** statement may be followed by an action statement. An action statement is executed if and when its **accept** statement succeeds. One of the **accept** statements may be replaced by either an **else** statement, which succeeds immediately if no other **accept** can succeed or, a **delay** statement, which succeeds if no other **accept** statement succeeds before completion of the specified period. The **else** statement is therefore equivalent to a **delay** statement with a zero time value. If more than one **entry** call is pending when the **select** statement is executed, selection between the **accept** statements is arbitrary.

```
task client is.
    ..
end client;

task body client is
begin
    select
        server.service_i();
            action_i();
        ...
        or
        server.service_j();
            action_j();
        or
        delay <period>;
            altaction();
    end select;
end server;
```

Figure 62 The Select Statement (entry call)

The **select** statement may be transformed into the generic input function receivefrom($\mathcal{S}$, m, t) by mapping each **accept** statement onto a member client of the set $\mathcal{S}$ and the **delay** statement time value onto the time parameter t. The value returned by receivefrom($\mathcal{S}$, m, t) is mapped onto the selection of an $action_i()$ or the altaction().

A task may also use a **select** statement to time its entry calls as shown in Figure 62.

The mapping onto the generic input function receivefrom($\mathcal{S}$, m, t) in this case is not so clear, as the client performs no explicit input. However, the implementation of a rendezvous must employ some protocol using messages between the client and server as shown in Figure 63. The client timeout is then set on the *request* ***accepted*** message.

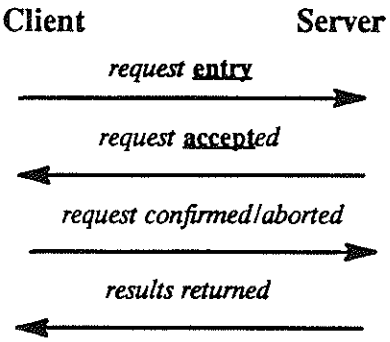

Figure 63 The Rendezvous Protocol

The **select** statement with entry calls may therefore be transformed into the generic input function by mapping the input of each *request* ***accepted*** message onto a member client of the set $\mathcal{S}$ and the **delay** statement time value onto the time parameter *t*. The value returned by receivefrom($\mathcal{S}$, m, t) is mapped onto the selection of an $action_i()$ or the altaction() (timeout).

7.4 Other Languages

In addition to OCCAM and ADA, we have also investigated the sources of non-determinism in Fault-Tolerant Concurrent C (FTCC) [68]. The message input-output constructs of this language are a mixture of those present in the two languages considered here, thus can be mapped onto the generic input function.

Concurrent programs are often composed of separately compiled modules written in a sequential language (e.g. C) which interact by calling communication primitives provided by an underlying operating system. The Unix system for example, provides a *select* primitive for message selection; its functionality is broadly similar to the select construct of ADA.

7.5 Summary

This section presented a number of programming constructs taken from current programming languages which are potential sources of non-determinism and showed how they could be mapped onto the *Generic Input Function*. Because state divergence can be prevented when using the *Generic Input Function*, as shown in Section 6.2.4, it can also be prevented when executing the example constructs. Such mappings could be performed by a pre-processor to allow the automatic distribution of a program written in a conventional programming language.

Chapter 8 : Software Architecture of a TMR Node

8.1 Introduction

This section presents a proposal for a software architecture to support the replicated processing system based on the architecture of Chapter 4. The design will incorporate the *order process* and *generic input function* as a mechanism for the prevention of state divergence. The Unix–like operating system called Helios [69] will be used to illustrate how the mechanisms discussed in this thesis can be implemented. We will assume that the application programs follow a modular, *object–oriented* style.

Helios supports the client/server model for the structuring of application programs [70], as described in Section 8.2. However, the amount of work involved in establishing and maintaining this model is not negligible. When applications are written in a high–level object–oriented language such as C++ [71], an abstract view of this model which hides the complexities of client/server creation and communication would be beneficial to the programmer. Such an abstraction can be implemented as a number of C++ class types from which application classes may be derived (see Appendix A, which gives the details of one such working implementation).

The client/server model also provides a convenient tool for the implementation of the various mechanisms described in this thesis. By incorporating a queue into the data part of the server (see Figure 64 and Appendix A), the structures of Figure 35 and Figure 44 may be implemented using a single queue_server type as described in the following sections. The *generic input function* of Section 6.2.4 becomes an additional member function of the server implementing VMQ. The system may then be used to prevent state divergence in application programs by performing a mapping of the application program onto the *generic input function* as described in Chapter 7.

8.2 The Helios Operating System

A Task as defined by Helios is a self-contained program unit which has been separately compiled and linked. A task may be multi-threaded but all threads share the same data space and are constrained to run on the same processor.

An application may be structured as a set of tasks (task force) which communicate using facilities provided by Helios. True concurrency may then be obtained by distributing the task force over several processors.

There are three distinct ways in which a task force may be structured; using pipes, using the Component Distribution Language (CDL) and using clients and servers. In general, the choice of method for a particular application must balance the flexibility of the solution against the complexity of its implementation.

Both pipes and CDL use streams (*stdin, stdout etc.*) for inter-task communication and produce static software architectures. Clients communicate with named servers using Helios primitives. So an application program using clients and servers provides a more dynamic software architecture.

Helios itself is based on the client/server model. Server tasks are provided by Helios to control access to system resources such as screens, keyboards and disks and usually reside on the processor closest to the resource that they manage. Application programs act as clients to these servers by sending requests and receiving replies (Remote Procedure Calls [70]) and may be located anywhere in the processor network. A client task may send requests to many servers during its lifetime, so the structure of the Helios+ application task force is continually changing under the direct control of the application. Fortunately, all the mechanisms used by Helios to provide the client/server framework are also made available to application task forces.

There are three main parts to a typical server as shown in Figure 64. The Active part corresponds to the main body of a conventional single-threaded task. It continues to

execute during the lifetime of the server. The Data part corresponds to the data space of the task. The Passive part which also has access to the Data Part distinguishes a server from any other task. Whenever a request is received from a new client, a new thread is spawned by the Helios dispatcher to deal with it. The code for the thread (the Passive part) is provided by the application but invoked automatically by Helios and may be thought of as a Remote Procedure Call (RPC) handler. In general, the passive part will continue to exist after completion of the RPC, remaining dormant (hence the name passive) until a further request is received from the same client. When the client has no further use for the server, the connection is broken and the RPC handler destroyed. If requests are received from a number of clients, a separate RPC handler is spawned for each client.

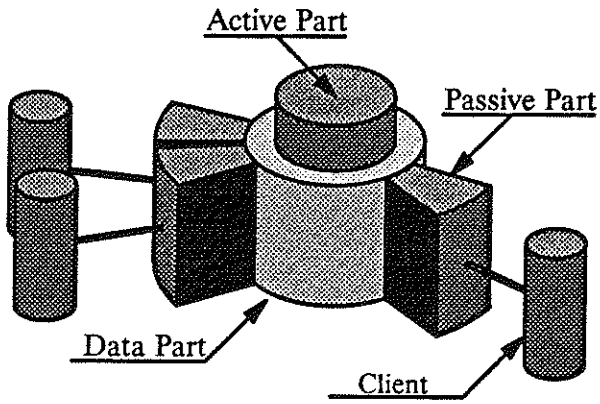

Figure 64 A Helios Server with Clients

The sequence of actions which comprise a typical client/server interaction are shown by Figure 65. The name of the new server (screen) is passed by the Active Part to the name server along with a port descriptor, then a dispatcher process is spawned to listen on that port. Thereafter, a locate request issued by a prospective client returns the server's port descriptor. On receipt of an open request, the dispatcher spawns a new Passive Part which returns a unique port descriptor to the client, to be used in all future requests. A remote procedure call is implemented using two messages; a request from client to server followed by a reply from server to client. When the client has no further use for the server, a special kill message causes the Passive Part to terminate.

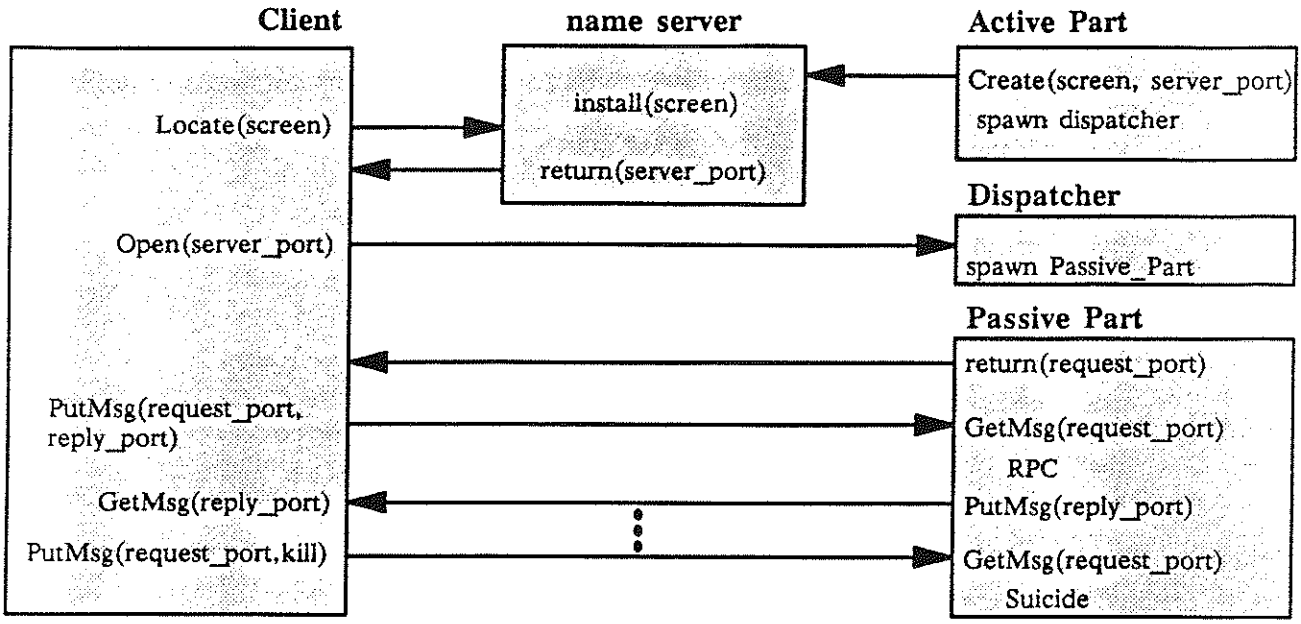

Figure 65 Client/Server Communication

8.3 Software Structure Overview

The software structure of Figure 35 may be represented by the block diagram of Figure 66 (intermediate nodes are not considered at this stage). The dual signature service (DSS) corresponds to the voter process, the message distribution service (MDS) to the destination receiver process and the order process service (OPS) to the order process.

When an application process wishes to make an RPC to a named service, it declares a queue_client. The queue_client establishes a connection to the local DSS. The local DSS establishes a connection to the remote MDS on the destination processor. The remote MDS is connected to the remote OPS. Finally the remote OPS establishes a client connection to the named service (queue_server).

The RPC is sent (in the form of a message) to the DSS, which generates a double-signed message, as described in Figure 39 (and Figure 36). The double signed message, into which the name of the service has been encoded, is sent to the MDS. The MDS distributes the message to its neighbours using the algorithm described in

Figure 40. One copy is sent to the OPS. The OPS performs the atomic broadcast algorithm of Figure 47, before decoding the name of the service and delivering the message (to the VMQ of the queue_server). A similar procedure must be followed to return the RPC reply to the client.

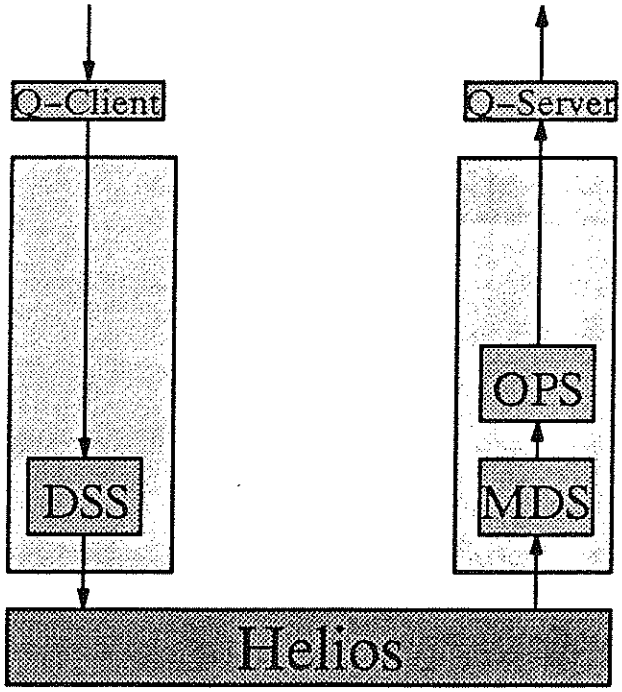

Figure 66 Software Block Diagram

All three services DSS, MDS and OPS require communication between neighbouring processors within a triad. They could simply communicate using Helios, but there are sound reasons why such an approach is undesirable. The powerful client/server connection described above, is not needed. A simpler, more efficient mechanism could be devised which takes advantage of the direct connection. This would improve the performance of the clock synchronization algorithm and hence the atomic broadcast, upon which the mechanisms used to implement the enhanced processing model of this thesis depend.

All intra-node message traffic will now be handled directly, without recourse to Helios. There must therefore be a neighbour communication service (NCS), as shown in Figure 67. The NCS will use the same client/server interface as the rest of the system

when communicating with the other services but will use its own methods for communicating within the triad on behalf of those services.

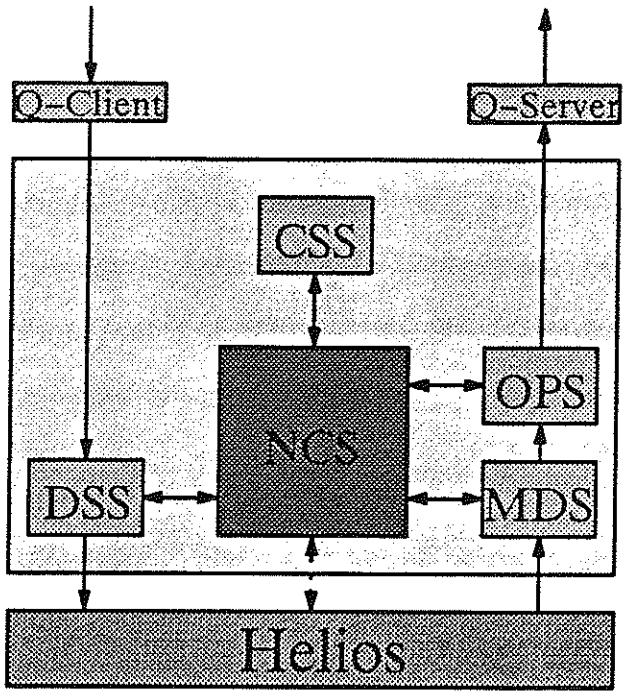

Figure 67 The Neighbour Communication Service

When an service wishes to send a message to a named service on a neighbouring processor within the triad, it declares a queue_client. The queue_client establishes a connection to the local NCS. The local NCS establishes a connection to the NCS on the neighbouring processor. Finally the neighbouring NCS establishes a client connection to the named service.

The message is sent to the local NCS, which appends the name of the service. The message is then sent to the neighbouring NCS which extracts the name and delivers the message to the service.

The encoding and decoding of the names of services will be performed by classes derived from the queue_client and queue_server classes, and bearing the same user interface. In this way, users of the facility will imagine that they are directly connected, and algorithms may be written without regard to actual locations.

There will also be a need for a fifth service, the clock synchronization service (CSS) which will be responsible for maintaining the synchronization of the processor clocks within the triad.

8.4 The Queue Server

When a task declares a service (an instance of a queue_server) (see Section A.4), the result may be represented by the functional diagram of Figure 68. Messages received from clients are placed at the back of the queue. The message at the front of the queue is removed by calling a *pop* operation on the queue_server object.

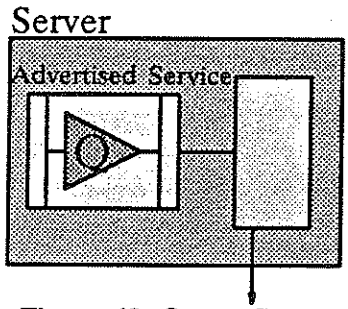

Figure 68 Queue Server

A task may declare more than one service (queue_server) as shown in Figure 69. A message is removed from a particular queue by invoking a pop operation on the appropriate queue_server object.

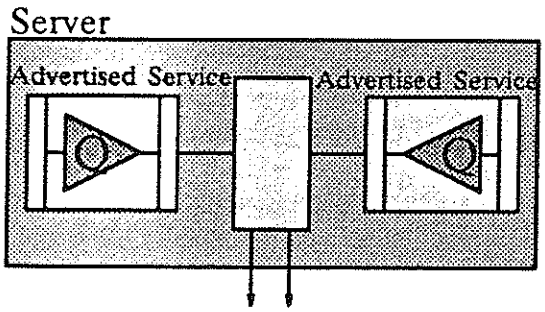

Figure 69 Double Queue Server

8.5 Building with Queue Servers

The block diagram of Figure 67 may now be redrawn as a complete functional diagram, as shown in Figure 70.

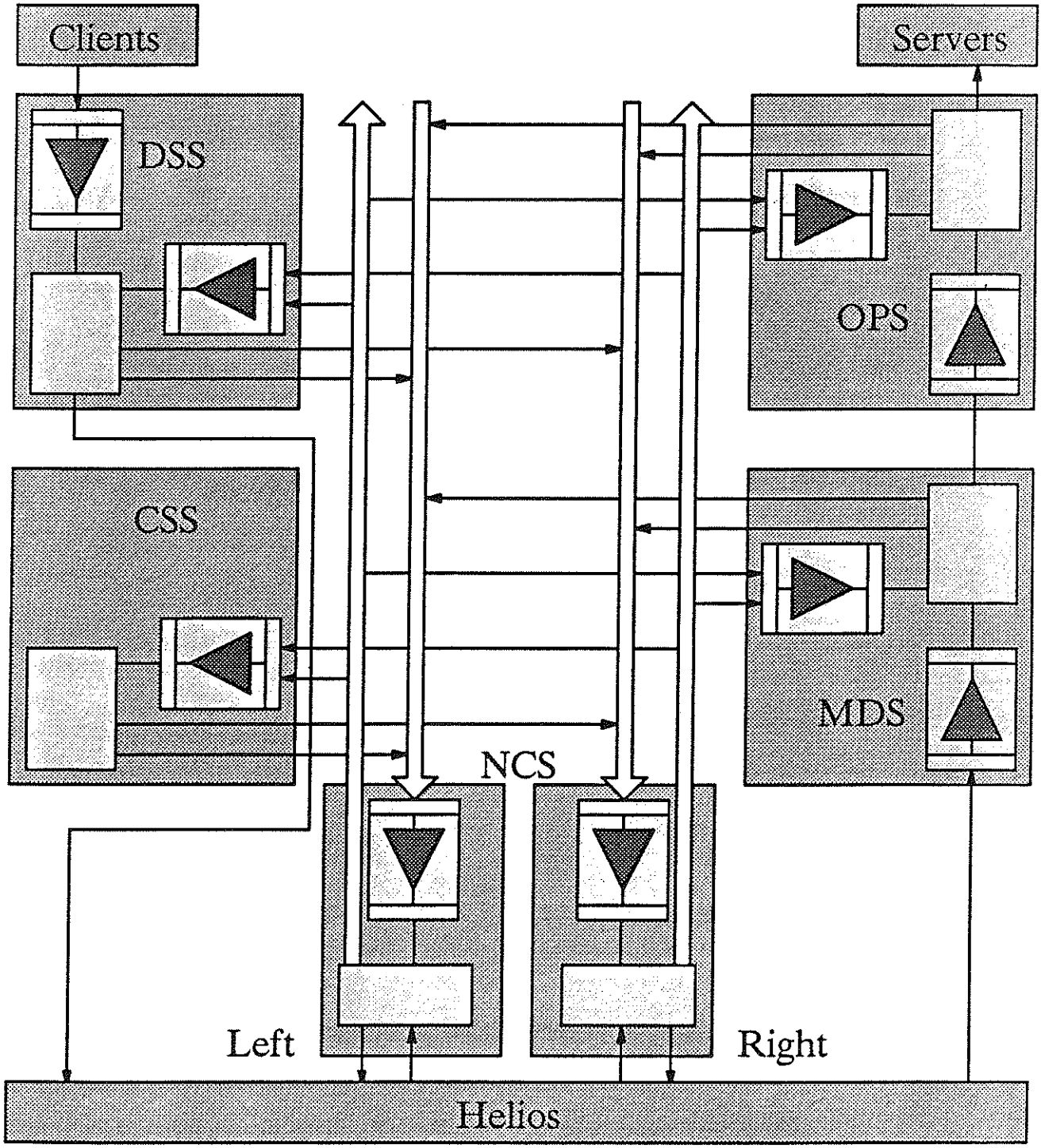

Figure 70 The Complete Functional Diagram

All services provide a queue for messages from their neighbouring counterparts (via NCS) and a client connection for sending messages to their neighbours' queues (via NCS). If we represent A uses B by A > B then:

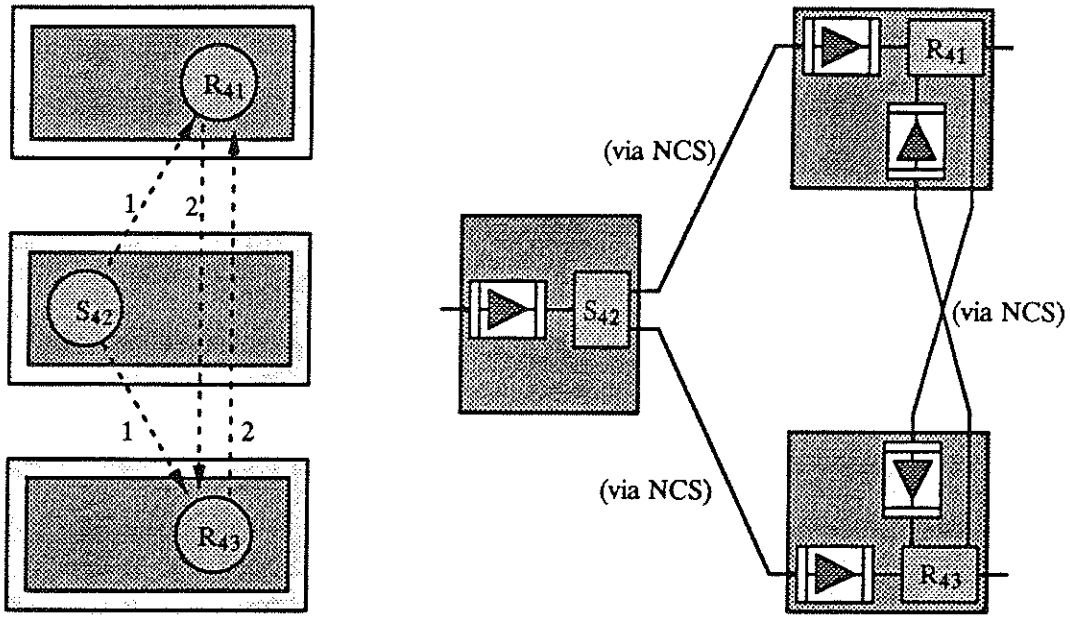

Figure 71 Order Process Implementation

service > NCS > NCS > *service*

All services (except CSS) provide a queue for messages from their clients and a client connection for sending messages to the queues of the services they use. If we represent A uses B by A > B then:

application_client > DSS > MDS > OPS > *application_service*

It may be desirable when implementing some of the services to use multiple threads. In such a case, each thread could advertise its own service by declaring a queue_server. This would allow the service structure to be designed in a way which reflects its logical construction. For example, in the order process structure of Figure 44, reproduced in

Figure 71, the *start* and *relay* processes must communicate within the triad. A logical solution is to place a queue at each point of input.

Thus an arbitrary structure can be built using the queue_server as a building block thereby allowing the designer of the service to concentrate on algorithms without considering the intricacies of implementation.

8.6 Summary

This section has presented the proposed design of a TMR node. The major building–blocks, the client/server classes have been implemented and the *generic input function* has been tested under the assumption that order is available. A complete implementation is now being developed by a project group consisting of the author and several colleagues.

Chapter 9 : Concluding Remarks

This thesis began by introducing the *state machine* as a model for active replicated processing and showed how the *consistent message ordering* requirement of the replicated state machine could be achieved using a solution to the *interactive consistency problem*. Interactive consistency was then examined in more detail using the well-known *Byzantine Generals Problem* analogy and several solutions to the problem were presented. Expansion of the state machine model to incorporate *non-determinism* was considered, to show that consistent message ordering alone is not sufficient to prevent *state divergence* among process replicas.

A range of existing system architectures were then presented to show how they prevent state divergence by either using passive replication techniques, by constraining processes to be deterministic, by using hardware support to execute processes in lock-step or by executing agreement protocols.

The new architecture of a multi-processor distributed system was then proposed, to support active replicated processing using TMR techniques and tolerate the arbitrary failure of one processor per TMR node. Several important issues were addressed, such as the use of digital signatures and the detection of duplicate messages. A degenerate form of the node was developed to have *fail-silent* semantics.

The *order process* was introduced as a mechanism for preventing state divergence in processes which conform to the state machine model. Implementation of the order process was considered using protocols which tolerate various classes of failure.

The order process was then used to prevent state divergence in an *enhanced processing model* which incorporates more complex message selection criteria, as expressed by the generic input function, asynchronous external events, non-deterministic processing, semaphores and real-time clocks.

A number of programming languages were investigated for constructs which could be potential sources of non–determinism and thus cause state divergence among process replicas. It was then shown how such constructs could be mapped onto the *generic input function* for which a solution had already been presented.

This thesis has presented a generic mechanism (the order process and generic input function) for the prevention of state divergence when applying NMR techniques to achieve fault–tolerance in highly safety–critical distributed systems. The mechanism will support an enhanced processing model (superior to the state machine model) while tolerating various types of processor faults up to and including arbitrary (Byzantine) faults. This is a significant advance on existing systems which restrict either the processing model (as in SIFT [28]) or the fault model (as in Tandem). A particularly important enhancement to the processing model is the incorporation of asynchronous external events which makes the mechanism suitable for real–time systems such as railway signalling [79] and nuclear power plant control [38].

The same mechanisms may also be used with a degenerate form of the TMR node architecture to construct processors with fail–silent semantics [13] and other fail–controlled processor architectures [12], as discussed in Section 4.6. Such processors can be used as building blocks for architectures such as Delta–4 XPA.

The mechanism does not assume any particular type of processor interconnection network (unlike SIFT and FTMP) and does not require any special hardware (unlike Delta–4 which requires a fail–silent NAC). Thus the mechanism may be used to incorporate fault–tolerance into any existing non–replicated multiprocessor architecture.

The generic mechanism requires processor clock synchronization only between members of a triad (to implement the atomic broadcast), so there is no need for global clock synchronization (as in other systems such as SIFT) and the mechanism is therefore suitable for massively parallel systems. This is due largely to the use of logical

clocks to determine the *age* of messages and to the confinement of *atomic broadcast protocols* to within a triad of directly connected processors.

The confinement of *atomic broadcast protocols* also reduces their termination time Δ_t by reducing δ, the time taken to transmit a message between participating processors and reducing ϵ, the synchronization error between the clocks of those processors (see Section 5.3.4). Since the overheads introduced by the mechanism are proportional to Δ_t (see Section 6.7), the confinement also greatly improves the mechanism's performance.

If the performance of the mechanism is insufficient for a particular application, then either the underlying fault model should be restricted, as in the Delta–4 system [46][47], thereby simplifying the *atomic broadcast protocol* and reducing Δ_t, or the computational model should be made deterministic, as in SIFT [17][19][28] or specialized hardware support should be used, as in FTMP [17][19][34], to limit the need for complex protocols.

If the *fail–silent* assumption for processes is made, then optimized solutions are possible; for example, the approach of forcing the decision of one replicate onto the others could be adopted, thereby preventing state divergence. Such an approach has been suggested for FTCC [72] and the *leader–follower* model for DELTA–4 XPA [50][51][52]. However, when the abstraction of *fail–silence* is implemented using *fail–arbitrary* processors (as in Sequoia [24][25][27] and Stratus [15][16][18]), the mechanism must be re–introduced (see Section 4.6). The current practice is to employ specialized hardware solutions. The fail–silent architecture made possible using the mechanisms of this thesis provides an alternative approach.

One possible development of the architecture proposed in this thesis is the incorporation of hardware support to drastically reduce Δ_t and hence greatly improve system performance. This hardware need not be complex as clock synchronization and atomic broadcast take place only within the confines of the processor triad. Taken to

the extreme, the processors of a triad could be tightly synchronized to execute processes in *lock–step* (as in FTMP) and asynchronous events synchronized to the processor instruction stream. Note however, that when applied to the system proposed in this thesis, only synchronization within the triad is required.

This thesis has presented a general–purpose software architecture and supporting mechanisms for building arbitrarily large replicated systems. More practical results are needed to evaluate the performance implications of the approach in an industrial setting. To this end, work is currently underway as part of the Delta–4 project.

Appendix A : A C++ Interface to Clients and Servers

A.1 Introduction

[10] The Helios Server of Figure 64 may be implemented in C++ as a task which declares an instance of a class. The Data Part is represented by the private class data members and the Passive Part by a private class member function. The Active Part is represented by the body of the task, accessing the Data Part via public class member functions.

In a typical application there will be a need for several distinct server types. Although their functions may vary widely, their basic structure is identical. It makes sense therefore to encapsulate this structure into a Base_Server class from which individual server class types may then be derived. Similarly, the mechanisms by which a client may connect to and communicate with a server may be encapsulated within a Base_Client class.

A.2 *The Base Classes*

The base classes are responsible for making, maintaining and breaking the client/server connection. The interface between the base classes and the derived classes consists of a single well-defined operation in each case (RPC_call,

```
typedef struct {
    unsigned short    DataSize;
    unsigned char     ContSize;
    word              *Control;
    byte              *Data;
    } Message_Block;
```

Figure 72 Message_Block

RPC_handler) with a consistent parameter (message block, of Figure 72). The derived classes are responsible for the mapping of individual procedure calls onto this

interface). In addition the derived server provides the target procedures which implement the server passive part of Figure 64.

```
class Base_Server
{    typedef struct UserDispatchInfo {
         DispatchInf   info;
         Base_Server  *that;};
     ObjNode           objnode;
     NameInfo          nameinfo;
     UserDispatchInfo  userinfo;
     struct Object     *server_obj;
public:
     Base_Server(char *);
     ~Base_Server();
     void do_open(ServInfo *);
     virtual void    RPC_handler
         (Message_Block *) = 0;
     virtual Message_Block
         *give_block() = 0;}
```

Figure 73 The Base_Server Declaration

The Base_Server class declaration is shown in Figure 73. The class data elements comprise all that is necessary to maintain the server and are hidden from the derived class. The public interface specifies that the derived class must supply two functions; the RPC_handler and give_block. The function do_open must be public for reasons explained later in this section.

To establish a server, two actions must be performed. Firstly, a dispatcher process must be created to spawn a new Passive Part process on receipt of an Open request from each new client. Secondly, the server must identify itself to the Helios name server so that it may be located by a prospective client. This may be accomplished by the class constructor as shown in Figure 74.

Consider first the action of identification. The server name, as passed to the constructor, is entered into the Helios name table by a call to Create. One of the parameters to Create is a NameInfo structure. One of the elements of NameInfo is a port descriptor called NamePort. This port descriptor, which corresponds to server_port in Figure 65, is passed back to the client in response to a Locate server request. The same

```
Base_Server::Base_Server(char *name)
{
    /*************************************/
    /*  Prepare information for the dispatcher */
    /*************************************/
    InitNode(&objnode, &(name[0]), Type_File, 0,
                                    DefFileMatrix);
    DispatchInfo *info     = (DispatchInfo *)&userinfo;
    info->Root             = (DirNode *) &objnode;
    info->ReqPort          = NewPort();
    info->SubSys           = SS_NullDevice;
    info->ParentName       = NULL;
    info->PrivateProtocol.Fn    = NULL;
    info->PrivateProtocol.StackSize = 0;
    for (int i=0; i<IOCFns; i++)
    {   info->Fntab[i].Fn = (VoidFnPtr)InvalidFn;
        info->Fntab[i].StackSize    = 2000;
    };
    info->Fntab[0].Fn      = (VoidFnPtr)Inter_do_open;
    info->Fntab[0].StackSize   = 5000;
    info->Fntab[2].Fn      = (VoidFnPtr)DoLocate;
    info->Fntab[3].Fn      = (VoidFnPtr)DoObjInfo;
    userinfo.that          = this;
    Fork(2000, (VoidFnPtr)Dispatch, 4, info);
    /*****************************************/
    /*  Enter the server into the Hellosname table */
    /*****************************************/
    nameinfo.NamePort   = info->ReqPort;
    nameinfo.Flags      = Flags_StripName;
    nameinfo.NameMatrix = DefFileMatrix;
    nameinfo.LoadData   = NULL;
    struct Object *sysroot =
                Locate(Null(struct Object), "/");
    server_obj = Create(sysroot, name, Type_Name,
                sizeof(NameInfo), (byte *)&nameinfo);
    Close((struct Stream *)sysroot);
}
Base_Server::~Base_Server()
{
    FreePort(nameinfo.NamePort);
    Delete(server_obj, "");
}
```

Figure 74 The Base_Server Class Constructor and Destructor

port descriptor is passed to the dispatcher process as the ReqPort element of the Info structure.

The dispatcher is created by forking a process. The function table within the Info structure tells the dispatcher which function to call on receipt of a particular type of request such as Open,Rename, Link, etc.. The entry for Open requests points to a function called Inter_do_open which must implement the Passive Part.

Interpretation of RPCs and generation of replies is the responsibility of the derived class. However, lower-level functions such as the reception, interpretation and

transmission of Helios messages may be handled by the Base_Server to contain unnecessary detail.

The Base_Server's Passive Part is implemented by the do_open class member function. Unfortunately, class member functions cannot be called directly by Helios (a 'C' world) as it has no way of initializing the implicit this pointer parameter. Instead, the this pointer is passed to inter_do_open (a 'C' function) explicitly by casting an additional

```
void inter_do_open(ServInfo *servinfo)
{
    ((((UserDispatchInfo *)servinfo->Info)
            ->that)->do_open)(servinfo);
}
```

Figure 75 Calling a Class Member Function

element to the Info structure (userinfo.that in Figure 74). This pointer can then be extracted from the servinfo parameter (by casting) to call the do_open member function of the appropriate class instance as shown in Figure 75. To the do_open function, this indirection is invisible.

The do_open function, as shown in Figure 76, begins by returning the request port descriptor to the client.

```
void    Base_Server::do_open(ServInfo *servinfo)
{
    /*****************************************/
    /*  Reply to 'open' command              */
    /*****************************************/
    MsgBuf  *r      = New(struct MsgBuf);
    FormOpenReply(r,servinfo->m,(struct ObjNode *)
                    userinfo.info.Root,Flags_Server,
                        servinfo->Pathname);
    Port request_port     = NewPort();
    r->mcb.MCBMsgHdr.Reply    = request_port;
    PutMsg(&r->mcb);
    Free(r); UnLockTarget( servinfo );
    /*****************************************/
    /*  Continue to direct RPC requests      */
    /*****************************************/
    MCB         mcb;
    word        function_code;
    Message_Block   *mb     = give_block();
    mcb.Timeout             = -1;
    mcb.Control             = mb->Control;
    mcb.Data                = mb->Data;
    mcb.MCBMsgHdr.Dest      = request_port;
    while ((GetMsg(&mcb) >= 0)
        && ((function_code = mcb.MCBMsgHdr.FnRc
                    & FG_Mask) != FG_Close))
    {   mb->ContSize= mcb.MCBMsgHdr.ContSize;
        mb->DataSize= mcb.MCBMsgHdr.DataSize;
        if (function_code == FG_Write)
        {   RPC_handler(mb);
            mcb.MCBMsgHdr.ContSize
                                = mb->ContSize;
            mcb.MCBMsgHdr.DataSize
                                = mb->DataSize;
            mcb.Control         = mb->Control;
            mcb.Data            = mb->Data;
            mcb.MCBMsgHdr.Dest
                        = mcb.MCBMsgHdr.Reply;
            PutMsg(&mcb);}
        mcb.MCBMsgHdr.Dest    = request_port;}
    FreePort(request_port);delete(mb->Control);
     delete(mb->Data);delete(mb);
    return;
}
```

Figure 76 The Base_Server Passive Part

It then continues to direct RPC requests (function_code == FG_Write) to the handler and returns RPC replies to the client until a terminating message is received (function_code == FG_Close). It then commits suicide. The call to get_block is to allow the size of the first message block to be determined by the derived class. Note that the message block which is returned by the handler containing the RPC reply is re-used to

store the following RPC request. It is the responsibility of the derived class to ensure that the size of this block is sufficient to contain the largest possible request.

The Base_Client class declaration is shown in Figure 77. The class data elements comprise all that is necessary to maintain the server connection and are hidden from the derived class. The public interface consists of a single function (RPC_call). When a client calls this function with a given message block the RPC_handler of the server is invoked with that block. Any reply generated by the server is returned to the client caller in the same block.

```
class Base_Client
{
    Port              reply_port;
    struct Stream     *server;
    MCB               mcb;
public:
    Base_Client(char *);
    ~Base_Client();
    void RPC_call(Message_Block *);
};
```

Figure 77 The Base_Client Declaration

To connect a client to a server, two actions are necessary. Firstly the server must be located, secondly an Open request must be sent to the server to spawn an RPC_handler. This may be accomplished using the class constructor as shown in Figure 78. The server port descriptor is contained within the severobj structure returned by the Locate request. The request port descriptor is contained within the server structure returned by the Open request. The latter is maintained as a class data element for the use of private member functions.

To break a client server connection, a special message must be sent to kill the server Passive Part for that particular client (function_code == FG_Close). This may be accomplished by the class destructor, also shown in Figure 78.

```
Base_Client::Base_Client(char *name)
{
    /*******************************/
    /*  Find the server.           */
    /*******************************/
    struct Object *sysroot;
    struct Object *serverobj;
    sysroot      = Locate(
                   Null(struct Object), "/");
    serverobj    = Locate(sysroot, name);
    Close((Stream *)sysroot);
    /*******************************/
    /*  Open a connection to the serve*/
    /*******************************/
    server       = Open(serverobj,
            Null(char), O_ReadWrite);
    reply_port   = NewPort();
    Close((Stream *)serverobj);
}

Base_Client::~Base_Client()
{
    InitMCB(&mcb, MsgHdr_Flags_preserve,
        server->Server, reply_port, FG_Close);
    mcb.MCBMsgHdr.DataSize     = 0;
    mcb.Data                   = NULL;
    mcb.MCBMsgHdr.ContSize     = 0;
    mcb.Control                = NULL;
    PutMsg(&mcb);
    Close(server);
    FreePort(reply_port);
}
```

Figure 78 The Base_Client Constructor and Destructor

Generation of an RPC_call and interpretation of the reply is the responsibility of the derived class. However, as with the Base_Server, low-level message passing may be handled by the Base_Client.

The Base_Client class provides an RPC_call function to send an RPC request using the FG_Write function_code then return the RPC reply to the caller, as shown in Figure 79. Note that the message block which contains the RPC request is re-used to store the RPC reply. It is the responsibility of the derived class to ensure that the size of this block is sufficient to contain the largest possible reply.

```
void Base_Client::RPC_call(Message_Block *mb)
{
    InitMCB(&mcb, MsgHdr_Flags_preserve,
            server->Server, reply_port, FG_Write);
    mcb.MCBMsgHdr.DataSize      = mb->DataSize;
    mcb.Data                    = mb->Data;
    mcb.MCBMsgHdr.ContSize      = mb->ContSize;
    mcb.Control                 = mb->Control;
    PutMsg(&mcb);
    mcb.MCBMsgHdr.Dest          = reply_port;
    GetMsg(&mcb);
    mb->DataSize    = mcb.MCBMsgHdr.DataSize;
    mb->Data        = mcb.Data;
    mb->ContSize    = mcb.MCBMsgHdr.ContSize;
    mb->Control     = mcb.Control;
}
```

Figure 79 Making a Remote Procedure Call

A.3 *The Derived Classes*

An implementation of an example Queue class pair is now presented to illustrate the use of the base classes.

```
typedef struct Queue_Obj
{           List        list;
            Semaphore   access;
            Semaphore   not_empty;};
```

Figure 80 The Queue Object

The Queue_Obj (Figure 80) employs the Helios structured types, List and Semaphore. The access semaphore is used to control access to the List to prevent a conflict between the active and passive parts of the server. The not_empty semaphore allows the active part to wait for data without consuming processor time. This technique should be used wherever a data structure must be shared between the active and passive parts of the server.

Both the Queue_Client and Queue_Server rely on the constructors and destructors of the base classes for server and connection creation and management. They need only create and manage their own data structures, as shown in Figure 81.

```
Queue_Server::Queue_Server
     (char *name):Base_Server(name)
{    InitList(&queue_obj.list);
     InitSemaphore(&queue_obj.access, 1);
     InitSemaphore(&queue_obj.not_empty, 0);}
Queue_Client::Queue_Client
     (char *name):Base_Client(name)
{    mb       = new(Message_Block);}
```

Figure 81 The Derived Class Constructors

The queue example implements two RPC's; reverse and push. They are encoded into a message by the client, as shown for push in Figure 82. They are then decoded by the

```
void Queue_Client::push(char *message)
{    word rpc= RPC_Push_Queue;
     mb->ContSize     = 1;
     mb->Control = &rpc;
     mb->DataSize     = strlen(message)+1;
     mb->Data     = message;
     RPC_call(mb);}
```

Figure 82 Encoding the RPC

server to vector control to the appropriate function as shown in Figure 83.

The RPC reply is generated by allowing the function to directly manipulate the data part of the message block via the message pointer it receives as a parameter.

```
void Queue_Server::RPC_handler(Message_Block *mb)
{   word    rpc = *(mb->Control);
    if (rpc == RPC_Reverse_Queue)
        reverse(mb->Data);
    else if (rpc == RPC_Push_Queue)
push(mb->Data);
}
typedef struct Entry
{   Node    node;
    char*message;};

void Queue_Server::push(char *message)
{   Entry *next_entry = new(struct Entry);
    int length          = strlen(message)+1;
    next_entry->message = new(char[length]);
    sprintf(next_entry->message,"%s",message);
    Wait(&queue_obj.access);
    AddHead(&queue_obj.list, &next_entry->node);
    Signal(&queue_obj.access);
    Signal(&queue_obj.not_empty);}

void Queue_Server::reverse(char *message)
{   int length    = strlen(message);
    for (int i = 0; i < length/2; i++)
    {
        char hold;
        hold = message[i];
        message[i] = message[length - i - 1];
        message[length-i-1] = hold;
    }}
```

Figure 83 Decoding and Performing the RPC

A pop function is provided for the use of the Active Part of the server (Figure 84). Note

```
char *Queue_Server::pop()
{   struct Entry  *next_entry;
    char      *message;
    Wait(&queue_obj.not_empty);
    Wait(&queue_obj.access);
    next_entry   =
        (struct Entry *) RemTail(&queue_obj.list);
    Signal(&queue_obj.access);
    message     = next_entry->message;
    Free(next_entry);
    return(message);}
```

Figure 84 Active Part Access Function

that in the case of a non-empty queue, the Active Part is forced to wait on a semaphore to avoid consuming processor time.

A.4 Using the C++ Interface

The following program segments show how client/server interface objects are used to communicate between processes. A Queue_Server is created by declaring an instance of the Queue_Server class as shown in Figure 85. The body of this program implements the server active part. Messages which have been placed in the queue in response to a push RPC are removed by the pop function and displayed on the screen. The name of the server (to be entered in the Helios name table) is derived from the command line used to invoke the program (argv[1]).

```
int main(int argc, char *argv[])
{
    if (argc != 2) {printf("illegal
            number of parameters\n");return(0);};
    Queue_Server    screen(argv[1]);
    char            *message;
    do
    {
        message     = screen.pop();
        printf("Message is %s\n",message);
    }
    while (strlen(message) > 1);
}
```

Figure 85 A Queue Server

A Queue_Client is created by declaring an instance of the Queue_Client class as shown in Figure 86. The name of the server to which the client must be attached is derived from the command line used to invoke the program (argv[1]). A string derived from the

```
int main(int argc, char *argv[])
{
    if (argc != 3) {printf("illegal
            number of parameters\n");return(0);};
    Queue_Client screen(argv[1]);
    char        *message = new(char[20]);
    sprintf(message, "%s", argv[2]);
    screen.push(message);
    screen.reverse(message);
    screen.push(message);
}
```

Figure 86 A Queue Client

second command-line parameter is sent to the server, reversed, then sent again. This

causes the string to be displayed twice on the screen associated with the server; once as normal and once reversed.

A.5 Summary

The base classes provide a convenient interface which encapsulates the application–independent aspects of Helios servers. The application programmer need only be concerned with the server data structures and their manipulation through RPCs as shown in the queue example of Section A.3. The derived classes show how the base classes may be used in a typical application. The queue_server (less the reverse function) in fact forms the basic building block for the implementation of the mechanisms described in this thesis (see Chapter 8). The derived classes are then used by declaring objects of type queue_client and queue_server.

Although at present the client/server structure is explicitly defined in the form of two separate programs (see Section A.4), it may in future be derived automatically. A standard C++ program will be processed by a stub generator [73] which will convert classes into servers and map invocations of class member functions onto calls to member functions of a corresponding client class. This would allow an arbitrary C++ program to be distributed across a network of processors running Helios.

References

[1] R. E. Lyons, W. Vanderkulk, "The Use of Triple–Modular Redundancy to Improve Computer Reliability", IBM Journal, April 1962, pp. 200–209.

[2] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals problem", ACM Transactions on Programming Languages and Systems, 4(2), July 1982, pp. 382–401.

[3] D. Dolev, "The Byzantine Generals Strike Again", Journal of Algorithms, Vol. 3, 1982, pp. 14–30.

[4] L.Lamport, "Using Time instead of Timeout in Fault Tolerant Distributed Systems", ACM TOPLAS, Vol. 6, No. 2, 1984, pp. 254–280.

[5] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, No. 7, July 1978.

[6] F.B. Schneider, "Abstractions for fault tolerance in distributed systems", Proceedings of IFIP 86 Congress, 1986, pp. 727–733, North Holland.

[7] F.B.Schneider, "The State Machine Approach", Section 8.2 in Distributed systems – Methods and Tools for specification: an Advanced course, Lecture Notes in Computer Science, Vol. 190, Springer Verlag, 1985.

[8] P.D.Ezhilchelvan, S.K.Shrivastava and A.Tully, "Constructing Replicated Systems Using Processors With Point to Point Communication Links", Proceedings of the 16th Annual Symposium on Computer Architecture, Jerusalem, June 1989, pp. 177–184.

[9] Alan Tully and Santosh K. Shrivastava. "Preventing State Divergence in Replicated Distributed Programs", Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, October 1990, pp. 104–113.

[10] A. Tully, "Distributed Programming on Tranputer Networks - An Object Oriented Interface to the Helios Operating System". Proceedings of the Second International Conference on the Applications of Transputers, Southampton, July 1990, pp. 296–302. IOS Press.

[11] P. Ezhilchelvan, S. K. Shrivastava, "A Characterization of Faults in Systems", Proceedings of the 5th Symposium on Reliable Distributed Systems, Los Angeles, California, January 1986.

[12] S.K. Shrivastava et al, "Fail–controlled processor architectures for distributed systems", Tech. Report, Computing Laboratory, University of Newcastle upon Tyne, 1990.

[13] R.D.Schlichting and F.B.Schneider, "Fail–stop Processors: an approach to designing fault–tolerant computing systems", ACM Transactions on Computing Systems, 1 (3), 1983, pp. 222–238.

[14] P. A. Lee, T. Anderson, "Fault–Tolerance Principles and Practice", Springer–Verlag, 1990, ISBN 0–387–82077–9, ISBN 3–211–82077–9.

[15] T. Anderson (Ed.), "Resilient Computing Systems", Collins, 1985, ISBN 0–00–383039–X.

[16] T. Anderson (Ed.), "Dependability of Resilient Computers", Blackwell Scientific Publications, Oxford, 1989.

[17] D. P. Seiwiorek, R. S. Swarz, "The Theory and Practice of Reliable System Design", Digital Press, 1982, ISBN 0–9232376–13–4.

[18] A. Avizienis, H. Kopetz, J. C. Laprie (Eds.), "The Evolution of Fault–Tolerant Computing", Springer–Verlag/Wien, 1987, ISBN 0–387–81941–X.

[19] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, "Definition and Analysis of Hardware and Software Fault–Tolerant Architectures", IEEE Computer, July 1990.

[20] D. A. Rennels, "Fault-Tolerant Computing - Concepts and Examples", IEEE Transactions on Computers, Vol. 33, No. 12, December 1984, pp. 1116–1129.

[21] S.K. Shrivastava, "Replicated distributed processing", Lecture notes in Computer Science, Vol. 248, 1987, pp. 325–337, Springer Verlag.

[22] J-C. Laprie, "Dependable Computing and Fault-Tolerance: Concepts and Terminology", Digest of papers, FTCS-15, Ann Arbor, Michigan, June 1985, pp. 2–11.

[23] D. Seiwiorek, "Fault-Tolerance in Commercial Computers", IEEE Computer, July 1990.

[24] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", Communications of the ACM, Vol. 34, No. 2, February 1991, pp. 57–78.

[25] F. Cristian, "Questions to Ask When Designing or Attempting to Understand a Fault-Tolerant Distributed System", Keynote Address, third Brazilian Conference on Fault-Tolerant Computing, Rio de Janeiro, September 1989.

[26] R. Horst, T. Chou, "The Hardware Architecture and Linear Expansion of Tandem Non-Stop Systems", Proceedings of the 12th Annual Symposium on Computer Architecture, June 1985.

[27] P. A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", IEEE Computer, February 1988, pp. 37–45.

[28] J.H. Wensley et al., "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", Proceedings of IEEE, 66, 10, October 1978, pp. 1240–1255.

[29] C. B. Weinstock, "SIFT: System Design and Implementation", Digest of papers, FTCS-10, Kyoto, Japan, June 1980, pp. 75–77.

[30] S. Frison, and J.H. Wensley "Interactive Consistency and its Impact on the Design of TMR Systems", Digest of papers, FTCS-12, Santa Monica, California, July 1982, pp. 228–233.

[31] H. Kopetz and W. Merker, "The architecture of Mars", Digest of papers, FTCS-15, Ann Arbor, Michigan, June 1985, pp. 274-276.

[32] H. Kopetz, et al, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", IEEE Micro, Vol. 9, No. 1, February 1989, pp. 25-40.

[33] I. M. MacLeod, "Data Consistency in Sensor-Based Distributed Computer Control Systems", ICDCS-4 San Francisco, California, May 1984, pp. 440-446.

[34] A. L. Hopkins, T. B Smith and J. H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", Proceedings of IEEE, 66, 10, October 1978, pp. 1221-1239.

[35] T. B. Smith, "Fault-Tolerant Processor Concepts and Operation", Digest of papers, FTCS-14, Kissimmee. Florida, June 1984, pp. 158-163.

[36] J. H. Lala, "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications", Digest of papers, FTCS-16, Vienna, Austria, July 1986, pp. 338-343.

[37] R.E. Harper, J.H. Lala and J.J. Deyst, "Fault tolerant processor architecture overview", Digest of papers, FTCS-18, Tokyo, Japan, June 1988, pp. 252-257.

[38] J.Lala, "A Byzantine resilient fault tolerant computer for nuclear power plant applications", Digest of Papers, FTCS-16, Vienna, July 1986, pp. 338-343.

[39] A. Whiteside, O. Tasar, L. Evans, D. Freedman, "Fault-Tolerant Multicomputer System for Control Applications", Digest of papers, FTCS-11, Portland, Maine, June 1981, pp. 286.

[40] C. J. Walter, R. M. Kieckhafer, A. M. Finn, "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems", Proceedings of IEEE Real-Time Systems Symposium, December 1985.

[41] R. M. Kieckhafer, "Task Reconfiguration in a Distributed Real-Time System", Proceedings of IEEE Real-Time Systems Symposium, December 1987.

[42] R. M. Kieckhafer, C. J. Walter, A. M. Finn, P. M. Thambidurai, "The MAFT Architecture for Distibuted Fault-Tolerance", IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 398–404.

[43] M. C. McElvaney, "Guaranteeing Deadlines in MAFT", Proceedings of IEEE Real-Time Systems Symposium, December 1988.

[44] R. M. Kieckhafer, "Fault-Tolerant Real-Time Task Scheduling in the MAFT Distributed System", 22nd Hawaii International Conference on System Sciences, January 1989.

[45] P. M. Thambidurai, A. M. Finn, R. M. Kieckhafer, C. J. Walter, "Clock Synchronization in MAFT", Digest of Papers, FTCS-19, Chicago, Illinois, June 1989, pp. 142–149.

[46] N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing", Digest of Papers, FTCS-19, Chicago, Illinois, June 1989, pp. 184–190.

[47] D. Powell et al., "The Delta-4 approach to dependability in open distributed computing systems", Digest of papers, FTCS-18, Tokyo, Japan, June 1988, pp. 246–251.

[48] P. Verissimo, L. Rodriguez, M. Baptista, "AMp: A Highly Parallel Atomic Multicast Protocol", Proceedings of ACM SIGCOM '89, Austin, Texas, September 1989.

[49] P. Verissimo, J. A. Marques, "Reliable Broadcast for Fault-Tolerance on Local computer Networks", Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, October 1990, pp. 54–63.

[50] P.A. Barrett et al, "The Delta-4 enhanced performance architecture (XPA)", Digest of Papers, FTCS-20, Newcastle upon Tyne, June 1990, pp. 481–488.

[51] D. T. Seaton, P. G. Bond, "Real-Time, Dependability and Distribution", Report of Esprit II P2252 Delta-4 Phase 3, Ref. W89.042/P2/C.

[52] D. T. Seaton, P. G. Bond, "Dependability and Ada", Report of Esprit II P2252 Delta-4 Phase 3, Ref. W89.043/D1/C.

[53] L. Lamport, P. M. Melliar-Smith, "Byzantine Clock Synchronization", Proceedings of 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, August 1984, pp. 68–84.

[54] J.Y Halpern, B. Simons, R. Strong and D. Dolev, "Fault tolerant clock synchronization", Proceedings of 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, August 1984, pp. 89–102.

[55] L. Lamport, P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", Journal of the ACM, Vol. 32, January 1985, pp. 52–78.

[56] T. K. Srikanth, S. Toueg, "Optimal Clock Synchronization", Proceedings of 4th Symposium on Principles of Distributed Computing, Minaki, Ontario, August 1985, pp. 71–86.

[57] O. Babaoglu and R. Drummond, "(Almost) no cost clock synchronization (preliminary version)", Digest of papers, FTCS-17, Michigan, June 1987, pp. 42–47.

[58] T. K. Srikanth, S. Toueg, "Optimal Clock Synchronization", Journal of the ACM, Vol. 34, No. 3, July 1987, pp. 626–645.

[59] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Approximate Clock Synchronization despite Omission and Performance Faults and Processor Joins", Digest of papers, FTCS-16, Vienna, Austria, July 1986, pp. 218–223.

[60] E.C. Cooper, "Replicated distributed programs", Proceedings of 10th Symposium on Operating System Principles, ACM Operating Systems Review, 19, 5, Dec. 1985, pp. 63–78.

[61] A.D. Birrell and B.J. Nelson, "Implementing remote procedure calls", ACM Transactions on Computer Systems, 2(1), February 1984, pp. 39–59.

[62] K. Echtle, "Fault masking and sequencing agreement by a voting protocol with low message number", Proceedings of 6th Symposium on reliability in distributed software and database systems, Williamsburg, March 1987, pp. 149–160.

[63] R. Rivest, A. Shamir and L. Adleman, "A method of obtaining digital signatures and public-key cryptosystems", Communications of the ACM, February 1978, pp. 120–126.

[64] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic broadcast: from simple message diffusion to Byzantine Agreement", Digest of papers, FTCS–15, Ann Arbor, Michigan, June 1985, pp. 200–206.

[65] Inmos Ltd, "Transputer Reference Manual", Prentice Hall, 1988.

[66] Inmos Ltd, "OCCAM 2 Reference Manual", Prentice Hall, 1988.

[67] I.C. Pyle, "The ADA programming language", Prentice Hall, 1981.

[68] R. Cmelik, N.K. Gehani and W.D. Roome, "Fault–tolerant concurrent C: a tool for writing fault–tolerant distributed programs", Digest of Papers, FTCS–18, Tokyo, Japan, pp. 56–61, June 1988.

[69] Perihelion Software Ltd. "The Helios Operating System". Prentice Hall International (UK) Ltd. 1989. ISBN 0–13–386004–3.

[70] A. D Birrell and B. J. Nelson. "Implementing Remote Procedure Calls", ACM Transactions on Computing Systems, Vol 2., No 1, 1984, pp. 39–59.

[71] Stanley B. Lippman. "C + + Primer", Addison–Wesley Publishing Company 1989. ISBN 0–201–16487–6.

[72] S. Arvelo and N.H. Gehani, "Replica consensus in Fault–tolerant concurrent C", Tech. Report, AT&T Bell Labs, 1989.

[73] Graham D Parrington. "Reliable Distributed Programming in C++: The Arjuna Approach", Digest of papers, Second Usenix C++ Conference, San Francisco, April 1990.

[74] N. Natarajan and J. Tang, "Synchronization of redundant computation in a distributed system", Proceedings of 6th Symposium on reliability in distibuted software and database systems, Williamsburg, March 1987, pp. 139–148.

[75] L. Mancini and S.K. Shrivastava, "Failure detection in replicated systems", Computing Laboratory, University of Newcastle Upon Tyne, Technical report no. 238, June 1987.

[76] L. Mancini and S.K. Shrivastava. "Exception handling in replicated systems with voting", Digest of papers, FTCS–16, Vienna, Austria, July 1986, pp. 384–389.

[77] T. Yoneda, T. Suzuoka, Y. Tohma, "Implementation of Interrupt Handler for Loosely–Synchronized TMR Systems", Digest of papers, FTCS–15, Ann Arbor, Michigan, June 1985, pp. 246–251.

[78] K.P. Berman, "Replication and fault tolerance in the ISIS system", Proceedings of 10th ACM Symposium on Operating System Principles, Washington, December 1985, pp. 79–86.

[79] N. Theuretzbacher, "VOTRICS: Voting triple modular computing system", Digest of papers, FTCS–16, Vienna, Austria, July 1986, pp. 144–150.

[80] L. Mancini, "Modular redundancy in a message passing system", IEEE Transactions on Software Engineering, January 1986, pp. 79–86.