Failure detection in replicated systems.

L. Mancini
S.K. Shrivastava
Failure detection in replicated systems

L. Mancini and S.K. Shrivastava

Abstract

Replicated processing with voting represents an attractive strategy for achieving reliability in real time systems. This technique permits N-Modular Redundant (NMR) nodes to be robust with respect to component failures. Three types of failures are identified in a replicated distributed system, and the paper discusses how these failures can be detected as exceptions by majority voters. These exceptions include sequence exceptions caused by out of order message processing and hardware exceptions caused by processor and node failures. In particular, standard and exceptional domains for majority voters are defined and implementation strategies for the detection and handling of the discussed exceptions are presented.

Series Editor: M.J. Elphick

© 1987 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

MANCINI, Luigi Vincenzo

Failure detection in replicated systems. [By] L. Mancini and S.K. Shrivastava


(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 238.)

Added entries

SHRIVASTAVA, Santosh Kumar
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Replicated processing with voting represents an attractive strategy for achieving reliability in real time systems. This technique permits N-Modular Redundant (NMR) nodes to be robust with respect to component failures. Three types of failures are identified in a replicated distributed system, and the paper discusses how these failures can be detected as exceptions by majority voters. These exceptions include sequence exceptions caused by out of order message processing and hardware exceptions caused by processor and node failures. In particular, standard and exceptional domains for majority voters are defined and implementation strategies for the detection and handling of the discussed exceptions are presented.

About the author

Mr. L. Mancini has been at the Computing Laboratory since May, 1985 as a Research Associate.

Professor S.K. Shrivastava joined the Computing Laboratory in August, 1975, where he is a professor.

Suggested keywords

DISTRIBUTED PROCESSING
EXCEPTION HANDLING
MAJORITY VOTING
REAL TIME SYSTEMS
RECONFIGURATION
RELIABILITY
REPLICATED PROCESSING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.887
Failure Detection in Replicated Systems
L. Mancini and S.K. Shrivastava
Computing Laboratory
The University
Newcastle upon Tyne, NE1 7RU, U.K.

ABSTRACT

Replicated processing with voting represents an attractive strategy for achieving reliability in real time systems. This technique permits N-Modular Redundant (NMR) nodes to be robust with respect to component failures. Three types of failures are identified in a replicated distributed system, and the paper discusses how these failures can be detected as exceptions by majority voters. These exceptions include sequence exceptions caused by out of order message processing and hardware exceptions caused by processor and node failures. In particular, standard and exceptional domains for majority voters are defined and implementation strategies for the detection and handling of the discussed exceptions are presented.

Index Terms - real time systems, majority voting, replicated processing, distributed processing, exception handling, reliability, reconfiguration.
1. Introduction

Modular redundancy in the form of replication of processing modules with majority voting is one of the best known techniques for tolerating failures of processing modules. In this paper we will explore the application of this form of redundancy to distributed systems and study one particular aspect - that of failure detection. This aspect is of particular relevance to highly reliable real time systems for two reasons: (1) time critical nature of real time systems often means that masking of failures by majority voting is the most appropriate treatment; (2) real time systems often need the capability of continuous operation in the presence of component failures - thus requiring the provision of on line reconfiguration to replace failed components. Given that replication of processing is desirable for achieving reliability in real time systems, detection of failures becomes an important task.

We consider our system to be composed of a number of nodes fully connected by means of redundant communication channels. A node will represent a functional processing module, composed of a number of processors and voters in a classical NMR (N-Modular Redundant) configuration. For such a distributed system, it will be shown that the traditional majority voting algorithm can be supplemented by exception handling algorithms to detect certain types of failures. The provision of exception detection and its treatment is the main subject of the paper.

The paper is structured as follows. In Section 2 we present a distributed system architecture for replicated processing. In Section 3 we describe failures in replicated systems; next we discuss principles of exception handling and apply those principles, in Section 5, to voting algorithms for the detection of exceptions during voting. Section 6 contains methods of treating those exceptions in NMR nodes. Conclusions from our study are presented in the last section. This paper is a revised and extended version of work reported earlier [MaSh86].

2. Distributed replicated processing architecture

The technique of replicated processing with majority voting has found widespread usage in systems requiring high operational reliability (e.g. space shuttle [SkI76]). A natural extension of a traditional single node system is a distributed system composed of a number of such nodes, as exemplified by the SIFT system [Wen78]. We will assume a functionally distributed system where each node performs a specific system function (in an avionics system, such functions could be sensor related processing, flight-path related processing and so forth). The overall architecture envisaged is depicted in Fig. 1, which shows triple modular redundant (TMR) nodes connected by a triplicated bus system (generalization to the corresponding NMR architecture is straightforward). An NMR node consists of N modules - each module being capable of performing processing and voting. To be specific, we will assume a module to be a single processor, such that voting is performed by the software that runs on that processor. Software implemented voting will enable us to exploit exception handling techniques for detecting failures. Each specific system
task will require processing at a subset of the nodes in the system. Suppose some particular task needs processing at nodes \( N_i \) and \( N_j \), in that order. Then, the three processors of \( N_i \) will perform the required processing and each processor of \( N_i \) will send triplicated messages to node \( N_j \) for subsequent processing. The voter of a processor \( P_k \) at \( N_j \) will vote the incoming messages, and pass on the voted message for processing to the software at \( P_k \) performing the processing function. Because we are assuming a fully connected system, (any node can send messages to any other node directly), nodes \( N_i \) and \( N_j \) can be regarded as logically connected as shown in Fig. 2. We will assume that the distributed system is performing many tasks concurrently. So, each node can receive messages from many other nodes for processing. For this reason, each processor of a node maintains a queue of voted messages (vmqs in Fig. 2). The main software components of a processor are thus a voter \( (V_i) \), voted message queue \( (vmq_i) \), and a task process \( (p_i) \) whose function is to pick up the message at the head of \( vmq_i \) and perform the required processing and send the result message (in triplicate) to some other node for subsequent processing.

In Fig. 2, a failure of a single processor in \( N_i \) can be masked by non-faulty voters of node \( N_j \) provided that: the task processes of all non-faulty processors of node \( N_i \) process identical messages in identical order and result messages from non-faulty processors of node \( N_i \) arrive uncorrupted at \( N_j \). Assuming that the only communication paths are those implied by the figure, no assumption about the behaviour of a failed component need be made. For the case just considered, arbitrary behaviour by the failed processor of \( N_i \) can be tolerated.

It will be assumed that each processor maintains some state information which affects subsequent processing. For example, if the task process \( p_k \) of \( N_j \) processes messages \( m_1 \) followed by \( m_2 \) and some other task process \( p_q \) of \( N_j \) processes \( m_3 \) followed by \( m_2 \), then results obtained by \( p_k \) and \( p_q \) for message \( m_2 \) need not to be identical.
It should be clear that to tolerate a maximum number of component failures within a node, the following sequencing condition must be met:

SEQ: All the task processes of non-faulty processors of an NMR node process voted messages in an identical order.

The sequencing condition is similar to the input order condition mentioned in [Sch85]. To understand the necessity for SEQ, consider the following specific case for TMR nodes (Fig. 2). Assume that for node \( N_j \), \( \text{vmq}_1 \) and \( \text{vmq}_2 \) are identical with messages \( m_2 \) and \( m_1 \) (at the head), and that \( \text{vmq}_3 \) contains messages \( m_3 \) and \( m_3 \) (at the head). So, \( p_1 \) and \( p_2 \) process \( m_1 \), while \( p_3 \) will process \( m_3 \). The subsequent processing by \( p_3 \) will in general produce different results from those of \( p_1 \) and \( p_2 \) and \( P_3 \) will act as a faulty processor. Thus, node \( N_j \) will not be able to tolerate any additional failures. However, if it is detected that \( P_3 \) is out of step and then \( \text{vmq}_3 \) is reordered to be identical to the queues of \( P_1 \) and \( P_2 \), the capability of tolerating processor failures can be reacquired.

The sequencing condition is particularly hard to meet in a concurrent processing environment, as can be appreciated by considering a simple example. Let \( N_k \) be an NMR node which can receive results from two different nodes \( N_i \) and \( N_j \) for two different tasks (Fig. 3). Suppose that \( N_i \) and \( N_j \) send their result messages at about the same time to \( N_k \), and that messages can experience
variable delays during transmission. It is thus possible that voters of $N_k$ receive messages in different order: this can cause voted messages to be queued in different order in vmqss.

In the following, three types of failures, and techniques for their detection and recovery in a replicated system will be described.

3. Failures in replicated systems

3.1. Failure types

The first type of failure to be considered is the sequencing failure: the violation of the sequencing condition in an NMR node. In addition to sequencing failures, there are two other types of failures that we will consider - processor failures and node failures. In order to precisely define these failures, we will first develop a fault model of the subsystem composed of a processor capable of sending messages to a voter (see Fig. 4). We will assume that a processor of a node such as $P_i$ and a voter of some node such as $V_k$ are connected by a logical communication link $L$ (L hides the details of actual communication path and network protocols used in a given implementation). We will regard a processor-link combination as a composite component $C_i$ with the following correct behaviour: let $m_j$ be the message at the input of $C_i$; then $C_i$ delivers a processed message $o_j$ such that $o_j$ is the expected output for the message $m_j$. Any violation of the above behaviour will be termed a processor failure relative to the voter. Referring to Fig. 4, we say that $P_i$ has failed relative to $V_k$.

A voter uses the majority voting technique to determine what constitutes the expected output from the unexpected. A node failure represents the case when there are enough processor failures to prevent a voter from forming a majority vote. More precisely, let $P_1, ..., P_n$ be the processors of a
node, say $N_i$, sending messages to a voter $V_k$ (see Fig. 5). Assume that no sequence failure has

occurred in the composite component $C$ and that all correct components $C_i, 1 \leq i \leq n$, have identical messages at their respective inputs. Then, the node $N_i$ fails relative to $V_k$ if the number of processor failures relative to $V_k$ is greater than $\lceil n/2 \rceil$. 

- 5 -
The following three remarks on the fault model presented here are worth noting: (1) abstracting away the details of actual communication technique permits us to treat faulty behaviour of systems with different communication structures identically; (2) link failures appear as processor failures to a voter; and (3) processor and node failures have been defined with respect to a given voter; it is for example possible for communication failures to occur in such a manner that $V_1$ of $N_j$ (Fig. 2) cannot form a majority whilst $V_2$ and $V_3$ can - thus $N_i$ will fail relative to $V_1$ but will appear correct to $V_2$ and $V_3$.

3.2. Failures as exceptions

One possible way of preventing sequencing failures is by employing an atomic message broadcast facility as discussed in [CASD86]. If all non-faulty processors use this facility for broadcasting their messages to their receivers, then all non-faulty voters of a node are guaranteed to receive messages in an identical order, thereby preventing the possibility of sequencing failures. In the absence of an atomic broadcast facility, we believe that there can be three different approaches for meeting the condition $SEQ$.

(i) Specific scheduling algorithms are designed to ensure that non-faulty voters of an NMR node insert voted messages in their vmbqs in an identical order [EzSh].

(ii) Non-faulty processors of an NMR node periodically execute an agreement protocol to ensure that vmbqs are identical [Sch85, Man86].

(iii) Occurrence of sequencing failures are detected as exceptions by voters and specific exception handlers are provided for recovering from such failures [MaSh86].

We draw the reader's attention to papers [EzSh, Man86] where algorithms are presented for the first two approaches. They have one feature in common: they prevent the occurrence of sequencing failures. In the third approach, sequencing failures are permitted but there is a provision for their detection and recovery.

If one assumes that in a given system the probability of sequencing failures is quite low, then the third approach becomes quite attractive. (The analogy with the treatment of deadlocks in an operating system is particularly instructive: if deadlocks occur rarely, prevention is often less attractive than detection with recovery). It might even be possible to design a system based on approach (i) or (ii) so as to reduce the probability of sequencing failures and then use approach (iii) for coping with them. Thus treating sequencing failures as exceptions appears to be a good strategy.

Another attractive reason for the introduction of exception handling is to pass valuable information to those parts of the system that are responsible for dynamically reconfiguring the system. Suppose that processor and node failures can be detected as exceptions by a voter. Then a voter can send such exceptions to that part of the system responsible for reconfiguration. The
design of reconfiguration system for processing such exception is a subject in its own right and not within the scope of this paper. We will however discuss some techniques of treating sequencing failures.

4. Principles of exception handling

In this section the basic concepts of exception handling are briefly introduced (for more details see [Cri82]). Let us consider a software system structured as a hierarchy of modules. When a module calls a procedure exported by a lower level module, then either the call terminates normally (the specified normal response is obtained) or a specified exceptional return is obtained, indicating that the called module was unable to provide the specified normal service. The caller can provide specific exception handlers for treating exceptional returns.

For a given module, one can define its standard domain (SD) such the execution of the module beginning in a state in SD terminates with normal intended results. If the module is executed with the initial state not in SD then the standard service may not be obtained (an exception will occur during the execution of the module). The set of such states is termed the exceptional domain (ED). It is the responsibility of the programmer designing a module to introduce run time checks in order to detect whether the module has been invoked in ED, and if so, to signal an exception to the caller.

In the following section the standard and three exceptional domains for a voter will be defined.

5. Exception detection during voting

We assume that the system receives requests for processing from the environment. Each request consists of \( n \) copies of a message encoding the processing requirements. A request message \( m_i \) is the tuple \(< t_i, r_i >\) where \( r_i \) is the request (or data) part of \( m_i \) containing the processing requirements, and \( t_i \) is a unique timestamp (a monotonic sequence number). As \( m_i \) passes from a node to node, only the data part gets modified, that is, a processor processing a request \( m_i \) will produce results which are placed in the data part before the results are sent to the following node. We assume that a processor (or voter) receiving a message can obtain the identity of the sender of the message. As a consequence of this assumption, a non-faulty receiver is in a position to send a response to the sender of a message.

A voter maintains a pool of messages received from other NMR nodes. For the sake of brevity, we will gloss over the details of the protocol required for collecting messages from the processors of a node. We will assume that, whenever a majority of messages from a given NMR node are present in the pool, the voter removes these messages from the pool and constructs an input n-tuple \(< in_1, \ldots, in_n >\) for voting (where \( n \) is the degree of replication). In particular, consider that the \( n \)-tuple is to be formed for messages received from an NMR node \( N_i \), then, if a message is present from processor \( j \), \( 1 \leq j \leq n \), of \( N_i \), \( in_j \) will contain this message, otherwise \( in_j \) is set to null.
Formally, let $M$ represent the set of all non-null messages. $M$ consists of the set of tuples of the form $<t_i, r_i> \in M$; the input domain $D_n$ of a voter is then given by:

$$D_n = \forall i: 1 \leq i \leq n : \text{in}_i \in (M \cup \{null\}) \land |\text{in}_i \land \text{null}| > \ln/2J$$

where, $n$ is the degree of replication.

The domain $D_n$ can be divided into a standard and three exceptional domains. In order to formally state these domains of a voter, the following two operations on $\text{in}_i$ are defined:

$$\text{in}_i.t = \text{IF } \text{in}_i = \text{null } \text{THEN } t_i \text{ ELSE null}$$

$$\text{in}_i.r = \text{IF } \text{in}_i = \text{null } \text{THEN } r_i \text{ ELSE null}$$

In addition, we use a function equal that returns the maximum number of non-null equal values in an $n$-tuple.

The characteristic predicate of the standard domain (SD) of the voter is:

$$SD = D_n \land (\forall i, j: 1 \leq i, j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal}(\text{in}_1.r, \ldots, \text{in}_n.r) = n$$

SD constraints the input $n$-tuple to contain only non-null identical messages.

The voter is invoked in the input exceptional domain (IED) if the following predicate is true:

$$\text{IED} = D_n \land (\exists i, j: 1 \leq i, j \leq n : \text{in}_i, \text{in}_j \in M \land \text{in}_i.t = \text{in}_j.t)$$

That is, there is a disagreement among the timestamps of non-null inputs. This may mean that the NMR node supplying these inputs has suffered a sequencing failure.

A voter is invoked in the hardware exceptional domain (HED) if the $n$-tuple has the following property:

$$\text{HED} = D_n \land (\forall i, j: 1 \leq i, j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \ln/2J < \text{equal}(\text{in}_1.r, \ldots, \text{in}_n.r) < n$$

That is, all non-null messages present for voting have identical timestamps and only a majority of those messages is in agreement. This situation can mean that the NMR node supplying the messages has suffered processor failures (processor failures have occurred, producing results that do not agree with the outputs of correct processors).

Finally, a voter is invoked in the fail exceptional domain (FED) if the following predicate is true:

$$\text{FED} = D_n \land (\forall i, j: 1 \leq i, j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal}(\text{in}_1.r, \ldots, \text{in}_n.r) \leq \ln/2J$$

That is, all the messages present for voting have the same timestamp, but no majority voting is possible. This represents a failure in reaching a majority vote and indicates a node failure that cannot be masked.
**Theorem 1.** The domain of the voter $V_k$ is given by the union of the standard domain SD and the three exceptional domains IED, HED, FED:

$$D_n = SD \cup IED \cup HED \cup FED$$

**Proof:** see the appendix.

The various domains are depicted in Fig. 6. The theorem allows us to state that if a non-faulty voter is capable of detecting the exceptions IED, HED, and FED, then it can provide a *total service* consisting of either normal responses or specified exceptional responses.

![Diagram](image)

**Fig. 6.** Partitioning of a voter domain

Let an NMR node $N_i$ send its outputs to an NMR node $N_j$ (Fig. 2). The algorithm for the voters of $N_j$ will then be described. A non-faulty voter $V_k$ of $N_j$ can simultaneously produce several outputs; for example, a signal and a voted output - the former for the sender node $N_i$, and the latter for $vmq_k$ of $N_j$. From the predicates given for the four domains, it is possible to device run time checks to determine in which domain a voter has been invoked.
Voting algorithm for voter $V_k$

DO

construct the input n-tuple for $N_i$;
cancel timer associated with $N_i$ if set;
IF input $\in SD \rightarrow$ insert the voted message in $vmq_k$;
    send \textit{successful voting} message to all the
    processors of node $N_i$;
    retry[i] := 0;
□ input $\in HED \rightarrow$ insert the voted message in $vmq_k$;
    send \textit{successful voting} message to all the
    processors of node $N_i$;
    construct a list of processors whose messages
    are not the same as the voted message;
    send a \textit{processor failure} message with
    this list to the reconfiguration subsystem;
    retry[i] := 0;
□ input $\in IED \rightarrow$ retry[i] := retry[i] + 1;
    IF retry[i] $\leq$ max-retry $\rightarrow$
        send a \textit{sequence exception}
        message to all the processors of $N_i$;
        % expect a new n-tuple from $N_i$ %
        set the timer for $N_i$;
        repeat the Voting algorithm;
    WHEN timeout expires
        % asynchronous timing exception indicating
        that no new input n-tuple could be formed for $N_i$%
    DO
        vote using the previous input n-tuple;
        IF majority voting possible $\rightarrow$
        % convert the domain to HED %
        set all the disagreeing messages in the n-tuple
        to the value \textit{Null};
        repeat the Voting algorithm with this n-tuple
    □ no majority present $\rightarrow$
        % treat as a node failure %
        send a $N_i$ \textit{fail} message to the
        reconfiguration subsystem;
        retry[i] := 0
    FI
□ retry[i] $>$ max-retry $\rightarrow$
    IF majority voting possible $\rightarrow$
        % convert the domain to HED %
        set all the disagreeing messages in the n-tuple
        to the value \textit{Null};
        repeat the Voting algorithm with this n-tuple
    □ no majority present $\rightarrow$
        % treat as a node failure %
        send a $N_i$ \textit{fail} message to the
        reconfiguration subsystem;
        retry[i] := 0
    FI
□ input $\in FED \rightarrow$ send a $N_i$ \textit{fail} message to the
    reconfiguration subsystem;
        retry[i] := 0
FI
OD
If the voter is invoked in its standard domain, SD, then it queues the voted message in \texttt{vmqk} and sends a successful voting message to all of the senders. The actions of the voter when it is invoked in HED is similar, except a processor failure message is sent to the reconfiguration system. The actions of the voter when it is invoked in IED are more complex. This is because, it must first be determined whether it is a genuine sequence failure or a failure caused by one or more failed processors (e.g. a faulty processor could have sent a result message with a modified timestamp).

For this purpose, a voter maintains an integer array called \texttt{retry} (initially set to zero at node startup time), with the \textit{i}th element associated with node \texttt{N}_i. When \texttt{V}_k detects IED from \texttt{N}_i, it signals an IED exception to \texttt{N}_i, and sets a timer. There are three mutually exclusive outcomes: (1) \texttt{N}_i performs recovery to recover from the sequence failure (see the next section) and re-sends the result messages; if recovery measures of \texttt{N}_i were successful, \texttt{V}_k will not be invoked in IED again; (2) \texttt{V}_k is invoked in IED again because recovery measures of \texttt{N}_i were not sufficient; or (3) the timer expires, indicating that no new n-tuple was made available by \texttt{N}_i. We assume that if \texttt{V}_k is repeatedly invoked in IED such that retry exceeds max-retry (where max-retry is a system constant), then processor failures or a node failure are indicated, and voting is either resumed by ignoring the relevant messages, or a \texttt{node fail} message is produced. Similar actions are undertaken if a timer expires.

The purpose of the retry mechanism associated with the IED exception is to be able to cope with the possibilities of arbitrary and malicious behaviour of failed components. This mechanism requires the voter to keep additional state information about the sender, and ensures that if faulty processors in node \texttt{N}_i cause a voter \texttt{V}_k to be invoked in IED, then the processor failures or the node (\texttt{N}_i) failure, as the case may be, will eventually be detected by \texttt{V}_k. However, if restrictive fault assumptions are made, for example that only omission failures can occur (a component either works as specified or fails by not producing any responses), then the voting algorithm can be simplified. Finally, the failure detection capability of a voter is worth noting: a voter only votes when at least a majority of messages from a node (say \texttt{N}_i) is available, so only under such a situation any failures of \texttt{N}_i will be detected.

The treatment of sequence exceptions by a node is presented in the next section.

6. Treatment of exceptions in NMR nodes

6.1. Use of backward error recovery

In terms of exception handling terminology, when a node \texttt{N}_i (Fig. 2) invokes a voter of node \texttt{N}_j, either a normal return is obtained or an exceptional return, indicating a sequence failure is obtained.

In order to cope with voter failures, the reply messages from voters themselves must be voted; such a voter will be referred to as an exception voter EV (Fig. 7). A task process, such as \texttt{p}_k of \texttt{N}_i, takes a copy of the message at the head of \texttt{vmqk}, performs the requisite computation, and sends n
copies of the result message to the following node \( N_j \). Process \( p_k \) then awaits a notification from \( EV_k \) (this synchronous processing ensures that all the non-faulty processors of \( N_i \) process exceptions in the same order as they are produced by \( N_j \)). If voter \( EV_k \) receives a majority of successful voting messages from \( N_j \), then there is no sequence failure and \( p_k \) discards the message at the head of the queue and picks up the next message for processing. If voter \( EV_k \) receives a majority of sequence exception messages, then it is known that some processors of \( N_i \) have processed out of sequence messages. Process \( p_k \) deals with this situation as follows: \( p_k \) checks if the message at the head of \( \text{vmq}_k \) has the smallest timestamp among those present in \( \text{vmq}_k \). If this is the case then \( p_k \) simply resends \( n \) copies of the result message to \( N_j \). If this is not the case then \( p_k \) performs backward recovery to undo the state changes produced by the computation performed for the most recently processed message of \( \text{vmq}_k \) and then reorders \( \text{vmq}_k \) to contain the earliest message at the head. This message is then normally processed. The voters of \( N_j \) will be in a position to vote again and respond to \( N_i \). It is worth noting that if \( N_i \) suffers a sequencing failure, then the recovery mechanism described above will ensure that all non-faulty processors of \( N_i \) will eventually have identical messages at the head of their respective \( \text{vmq} \). This is because the
reordering is performed on voted messages: if a message is in the queue of some non-faulty processor, then it will be either present or will eventually be present in the queues of all other non-faulty processors.

6.2. Recovery using input voting

The drawback of the approach presented above is that it requires a processor to be capable of performing backward error recovery. One possible way of avoiding the use of backward error recovery is to detect a sequence failure before the processing of a message by inserting a voter between a vmq and its task process. Such a voter receives its inputs from all the vmqs of that node, thus ensuring that all non-faulty processors of Nᵢ process messages in an identical order (Fig. 8).

![Diagram](image)

Fig. 8. Input voting

The function of input voter VSᵩ of node Nᵢ is to detect sequence exceptions. If VSᵩ is invoked in IED then it signals an exception Sᵩ; this causes the task process of processor Pᵩ to re-order its vmqᵩ as was mentioned in the previous sub-section.

It is worth noting that voters VHᵩ and VSᵩ together are responsible for detecting the three types of failures: VHᵩ detects processor and node failures and VSᵩ detects sequence failures. VHᵩ performs voting using only the data part of n-tuples while VSᵩ performs voting using only the timestamps of the n-tuple presented to it.
Theorem 2. The domain of the voter VHₖ is given by the union of the standard domain SD and the exceptional domains HED, FED:

\[ Dₙ = SD \cup HED \cup FED \]

Proof: see the appendix.

Theorem 3. The domain of the voter VSₖ is given by the union of the standard domain SD and the exceptional domain IED:

\[ Dₙ = SD \cup IED \]

Proof: see the appendix.

From theorem 2 we see that the first voter, such as VHₖ, is never invoked in IED, so its algorithm can be constructed from that given in Section 5 minus the retry mechanism and the code associated with the IED part.

Similarly, using theorem 3, we can construct the algorithm for the second voter, such as VSₖ, from the algorithm given in Section 5 minus the parts associated with HED and FED.

7. Concluding remarks

We have discussed how the function of majority voters can be enhanced to detect failures in replicated distributed systems. Three types of failures were identified and a rigorous treatment for detecting them as exceptions during majority voting was presented. Processor and node failure exceptions can be used by a reconfiguration subsystem for removing faulty system components. The presence of concurrency can introduce the possibility of sequencing failures in replicated systems. If sequencing failures are not detected, the ability of NMR nodes to mask processor failures is degraded. Two methods of handling sequencing exceptions have also been presented.

The main advantages of the failure detection technique presented here are summarized below.

(i) Sequencing failures are a special class of failures that could occur in a concurrent system caused by factors such as variable transmission delays in communication. Such failures can be treated by reordering of messages in vmqs, and no reconfiguration of system components is required. The paper has described how a voter can detect such failures.

(ii) It is possible for an NMR node to tolerate a sequencing failure and ln/2J processor failures simultaneously.

(iii) Even if preventive measures are employed for dealing with the possibilities of sequencing failures, the exception handling framework presented here can be utilized for the detection of processor and node failures.

As mentioned in Section 3.2, one can consider the following approach: use preventive measures to reduce the probability of sequencing failures and then employ the technique presented here for
coping with them. It is likely that this approach will improve the overall performance of the system, since algorithms based on average transmission and processing delays can be constructed (thus leaving the possibility of sequencing failures). The development and evaluation of such a scheme is left as a topic for further research.

We conclude this paper by drawing the reader's attention to some related research work in the area of replicated systems with voting. A formal specification of the correctness requirements for replicated systems is discussed in [MaKo86]. In [Man86], the author discusses the problem posed by non determinacy in application programs, and techniques of coping with it. In [EzSh], the authors discuss scheduling of real time processing requests in replicated systems. An interesting voting protocol which prevents sequencing failures is presented in [Ech87]. In [Coo85], the author describes a replicated procedure call based system for executing distributed replicated programs. The replicated procedure call facility could form an attractive basis for communication between a processor and remote voters. In [YSS83], the authors describe experimental work to evaluate the performance of software voters and show that voting frequency is the dominant factor in determining the voting overhead.

Acknowledgements

The authors are grateful to Klaus Echtle, Paul Ezhilchelvan and Brian Randell for their technical comments on the ideas presented here. This work has been supported by grants from the U.K. Science and Engineering Research Council and the Royal Signals and Radar Establishment of the Ministry of Defence.
APPENDIX

Before proving the three theorems stated in Sections 5 and 6.2, let us consider the following lemmas.

**Lemma 1.** Let \( P_i \) be a predicate of the form \( P_i = Q \land P'_i \), then for any predicate \( Q \) the following property is satisfied:

\[
\sum_{i=1}^{m} P'_i = \text{TRUE} \Rightarrow \sum_{i=1}^{m} P_i = Q
\]

**PROOF:**

\[
\sum_{i=1}^{m} P_i = \sum_{i=1}^{m} Q \land P'_i = Q \land \sum_{i=1}^{m} P'_i = Q \land \text{TRUE} = Q
\]

q.e.d.

**Lemma 2.** Let us state each of the SD, IED, HED, and FED in the form \( Q \land P'_i \) where \( Q = D_n \) and SD', IED', HED', and FED' equal respectively to the characteristic predicates of SD, IED, HED, and FED after \( D_n \) has been deleted; then \( \text{SD}' \lor \text{IED}' \lor \text{HED}' \lor \text{FED}' = \text{TRUE} \).

**PROOF:**

\[
\text{HED}' \lor \text{FED}' = \\
= (\forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \frac{\ln 2.1}{\ln 2.1} < \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) < n \\
\lor (\forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) \leq \frac{\ln 2.1}{\ln 2.1} \\
= (\forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) < n
\]

HED' \lor FED' \lor SD' =

\[
(\forall i,j: 1 \leq i,j \leq n \colon \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) < n \\
\lor (\forall i,j: 1 \leq i,j \leq n \colon \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) = n \\
= (\forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \land \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) \leq n
\]

Since, \( \text{equal(in}_1, \text{r}, \ldots, \text{in}_n, \text{r}) \leq n \), the above equation can be simplified to:

\[
\text{HED}' \lor \text{FED}' \lor \text{SD}' = \forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t
\]

Finally,

\[
\text{HED}' \lor \text{FED}' \lor \text{SD}' \lor \text{IED}' = \\
= (\forall i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \Rightarrow \text{in}_i.t = \text{in}_j.t) \\
\lor (\exists i,j: 1 \leq i,j \leq n : \text{in}_i, \text{in}_j \in M \land \text{in}_i.t = \text{in}_j.t)
\]

= \text{TRUE}

where we have made use of the following two laws:
(1) \( \forall i: E_i = \neg (\exists i: \neg E_i) \)
(2) \( \neg (P \Rightarrow Q) = P \land \neg Q \)
q.e.d.

**Theorem 1.** The domain of the voter \( V_k \) is given by the union of the standard domain SD and the exceptional domains IED, HED, FED:

\[
D_n = SD \cup IED \cup HED \cup FED
\]

**PROOF.** The proof follows from Lemma 2 and Lemma 1.
q.e.d.

**Theorem 2.** The domain of the voter \( VH_k \) is given by the union of the standard domain SD and the exceptional domains HED, and FED:

\[
D_n = SD \cup HED \cup FED
\]

**PROOF.**

\( VH_k \) does not deal with sequencing failures. Its input domain \( D_n \) is defined by the same characteristic predicate given in Section 5 where \( M \) consists of the set of tuples of the form \(<\text{void}, r> \in M\), \( r \) is the request part of the message and \( \text{void} \) specifies that no timestamp is present.

In order to state the various domains of \( VH_k \), let us consider the partition of the domain \( D_n \) obtained by applying the same characteristic predicates given in Section 5. Since the timestamp field is always equal to the value \( \text{void} \), the domain IED is an empty set - its characteristic predicate is \( IED = D_n \land \text{FALSE} = \text{FALSE} \). We are then left only with three domains, namely SD, HED, FED. It is worth noting that \( SD' \lor HED' \lor FED' = \text{TRUE} \), because (1) from Lemma 2, \( SD' \lor IED' \lor HED' \lor FED' = \text{TRUE} \), and (2) \( IED' = \text{FALSE} \). Since Lemma 1 holds for any predicate \( Q \), and \( SD' \lor HED' \lor FED' = \text{TRUE} \), it follows that \( D_n = SD \cup HED \cup FED \).
q.e.d.

**Theorem 3.** The domain of the voter \( VS_k \) is given by the union of the standard domain SD and the exceptional domain IED:

\[
D_n = SD \cup IED
\]

**PROOF:**

In this case \( VS_k \) deals only with sequencing failures. Its input domain \( D_n \) is defined by the same characteristic predicate given in Section 5 where \( M \) consists of the set of tuples of the form...
\(<t, \text{void}> \in M, t\) is the timestamp of the message, and \(\text{void}\) specifies that no request part is present. It can be easily shown that the domains \(\text{HED}\) and \(\text{FED}\) are empty sets. It is worth noting that \(\text{SD}' \lor \text{IED}' = \text{TRUE}\), because (1) from Lemma 2, \(\text{SD}' \lor \text{IED}' \lor \text{HED}' \lor \text{FED}' = \text{TRUE}\), and (2) \(\text{HED}' = \text{FED}' = \text{FALSE}\). Since Lemma 1 holds for any predicate \(Q\), and \(\text{SD}' \lor \text{IED}' = \text{TRUE}\) it follows that \(D_n = \text{SD} \cup \text{IED}\).

q.e.d.
References


