Replicated distributed processing.

S.K. Shrivastava

Abstract

Replicated processing with voting provides a powerful means of constructing highly reliable computing systems. We will consider a functionally distributed computing system intended for real-time applications, where each functional module - a node - has been configured in an NMR (N-modular redundant) fashion. Such a system receives processing requests from 'actuators' (the entities that demand services) that require distributed processing at various nodes. The paper will discuss various approaches to scheduling computations to ensure that each processor of an NMR node processes input messages in an identical order. The concept of exception handling for voters will be developed to detect failures in the system.

This paper was presented at the IBM Europe Institute, 18th - 22nd August, 1986, Oberlech, Austria.

Series Editor: M.J. Elphick

© 1986 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

SHRIVASTAVA, Santosh Kumar


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1986.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 222.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Laboratory, Technical Report Series. 222.

Abstract

Replicated processing with voting provides a powerful means of constructing highly reliable computing systems. We will consider a functionally distributed computing system intended for real-time applications, where each functional module - a node - has been configured in an NMR (N-modular redundant) fashion. Such a system receives processing requests from 'actuators' (the entities that demand services) that require distributed processing at various nodes. The paper will discuss various approaches to scheduling computations to ensure that each processor of an NMR node processes input messages in an identical order. The concept of exception handling for voters will be developed to detect failures in the system.

About the authors

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

Suggested keywords

BYZANTINE AGREEMENT
DISTRIBUTED SYSTEMS
EXCEPTION HANDLING
FAULT TOLERANCE
MESSAGE PASSING
REPLICATED PROCESSING
VOTING

Suggested classmarks (primary classmark underlined)

Dewey (18th): 001.64404
U.D.C. 519.687
1. Introduction

One of the great challenges facing computer scientists is to design and build computer systems that provide *guaranteed services* in the presence of a finite number of failures. We will assume - and this hardly needs any justification - that many applications require computer systems which 'closely' approximate ideal systems that never fail (since physical systems will eventually fail, we can only approximate the ideal). In order to be able to provide any kind of guarantee of service, one must precisely specify what kinds of, and how many, component failures a system is supposed to tolerate.

Suppose our system is constructed out of 'n' components (where a component can either be a hardware or a software module); then its reliability specification could be along the lines that *provided* there are 'm' or less component failures (where \( m, m < n \), characterises the redundancy in the system) *and* each failure is of an assumed type, *then* the system will function as specified. Note that in such a specification, failure mode assumptions for components need to be stated explicitly, since if a component failure occurs that is outside the failure mode assumptions made, then no guarantee of normal services can be given. For many components (e.g. a microprocessor, an operating system) it is often hard to predict all possible failure modes that could occur. Then there is no alternative but to make *minimal* (and if possible *no*) assumptions about failure modes of such components and to design a system under the assumption that a failed component can behave in an *arbitrary* manner. In subsequent sections we will examine the consequences of this assumption on system design, but it is hoped that the reader will appreciate that design and construction of such systems is a remarkably difficult task and a substantial amount of research work is required. The purpose of this paper, which is tutorial in nature, is to acquaint the reader with some current research work in the area of highly reliable systems for real time processing.
2. On Redundancy and Byzantine Agreement

Let us first consider a fundamental reliability problem for interacting processes. Assume that a process A transmits a value to some other processes (B, C and D) and we wish to ensure that these processes do indeed receive the same value. Now, it is possible for A to fail in such a manner that - assuming that the value is binary - 'Yes' is sent to say B and C and 'No' to D (since we assume that a failed component can behave arbitrarily, we cannot discount this possibility). It is therefore necessary for processes B, C and D to take part in an agreement protocol to ensure that all of them decide on some common value. What if one or more of the processes taking part in the protocol also fail in the aforementioned manner? In a classic paper [1], Pease et al proved that, to tolerate m component failures, a consensus among non-faulty components can only be reached if the total number of components, n, is greater than 3m (n > 3m). Details of such protocols, popularly known as Byzantine agreement protocols [2] need not concern us here, it is enough to assume the existence of such protocols, and to note that they tend to be quite expensive in terms of message requirements.

Since the only way of providing fault tolerance is through the introduction of redundancy, we are often faced with the sort of problem discussed above. For example, B, C and D could be three processors that redundantly carry out the same task, in which case we must ensure that they receive the same input data before processing begins. A system intended for real time applications must provide prompt responses to service requests, so excessive use of agreement protocols must be avoided as far as possible, giving us a design rule for highly reliable real time systems:

A real time system should be structured so as to minimise the requirements for Byzantine agreement.

Let us now examine how redundancy can be introduced into a system. To be specific, let us consider how we can construct an "ideal" processor out of a
number (say n) of ordinary processors. There can be two radically different approaches:

1) **Fail Stop Processors (FSPs):** All of the n processors *interactively* (by making use of Byzantine protocols) maintain the following abstraction: any processor failure causes all of the non-faulty processors to stop (hence the name). We thus obtain an (almost) ideal processor - the FSP - that does not possess arbitrary failure modes, rather, it either works or simply stops [3]. FSPs can thus be used as building blocks for the construction of reliable systems.

2) **N Modular Redundant (NMR) Processors:** All the n processors *work in isolation* and use voting to *mask* outputs from faulty processors (see fig.1, where n = 3, giving us the well known triple modular redundant - TMR - system). It is assumed that a processor of an NMR node performs both voting and task processing functions (V and P) and that results from each P are sent to all the Vs of the next node where further processing will take place after voting.

In the system shown in fig. 1, the failure of a single V-P combination in node N_i can be masked by non-faulty voters of the subsequent node N_j provided that the non-faulty processors of N_i produced identical results which arrive at N_j uncorrupted. Assuming that the only communication paths are those implied by the figure, no assumption about the behaviour of failed processors need be made.

Out of the above two techniques, the NMR based approach appears more suitable, based on the following two observations: (i) FSPs do not mask failures, so application programs need recovery facilities which is not the case with the NMR approach; and (ii) FSPs require extensive use of Byzantine agreement protocols and require a rather complex internal structure [4].

In the rest of this paper we will concentrate on distributed systems composed out of NMR nodes. An NMR node must satisfy a synchronization
requirement, which is that all of its n processors be 'roughly in step' with each other. If this synchronization is to be achieved by the processors communicating with each other then we have to introduce extra communication paths in the system (not shown in the figure) and the need for Byzantine agreement surfaces once more. In the subsequent sections we will examine design requirements for distributed NMR systems processing replicated computations.

3. Distributed NMR System Architecture

We will assume a functionally distributed system architecture consisting of a number of NMR nodes fully connected by an N-redundant communication system. Each NMR node R_i, manages (or represents) some resource (e.g. sensors). The environment of the system consists of a set of initiators (entities that demand services from the system at arbitrary times) and a set of sensors (entities that monitor parameters of the environment, such as temperature, air pressure and so on). A service request from an initiator can give rise to a
processing activity involving several nodes; at any time there could be several such requests being processed by the system.

We will assume that any initiator request \( e_i \) requires distributed sequential processing: suppose the processing requirement for \( e_i \) is \( R_k; R_l; R_m \), indicating the sequence in which the request is to be processed at those nodes (at node \( R_k \) then at \( R_l \) and then at \( R_m \)). This request will be processed in our system as follows: replicated request \( e_i \) will be sent to the voters of \( R_k \); each processor of \( R_k \) will perform the processing and will forward replicated results to node \( R_l \) where the voting will take place before subsequent processing. Note that, for the sake of simplicity, we are restricting the processing of a request to be sequential; in particular this means that there are no synchronization requirements for the processing of any two requests \( e_i \) and \( e_j \) - other than the fact that each resource is to be used exclusively.

In general, a processor of an NMR node can receive several requests for processing, from other NMR nodes and initiators, which suggests the software architecture within a processor to be as depicted in fig. 2. A non-faulty processor maintains a pool of buffers for storing incoming messages. A voter performs voting as soon as it can form a majority on a given set of messages received from some node or an initiator. The voted messages are stored in a voted message pool. Some task scheduling policy (to be discussed shortly) is employed for selecting voted messages from the pool and queueing them in the voted message queue (VMQ) for processing. The messages in the VMQ are processed on a FCFS basis by the task process (\( P_i \) in fig.2).

We assume that \( P_i \) maintains some state which affects the execution of a task, and further that the execution of a task can modify the state. Assuming that all the non-faulty processes of a node have identical initial states before task processing begins, we require the following sequencing condition for an NMR node:
SEQ: all non-faulty task processes of an NMR node process voted messages in an identical order.

It will be the function of the task scheduler of a processor to satisfy the sequencing condition. Application level requirements may dictate some further constraints on selecting voted messages for processing - for example some messages could have a higher priority over others for processing (e.g. 'alarm' messages). Task scheduling is discussed at length elsewhere [5]; we will briefly address some approaches to meeting just the sequencing condition in the next section.

A violation of the sequencing condition in an NMR node will be termed a sequencing failure. It should be clear that sequencing failures reduce the failure masking capability of a node. Consider for example the situation depicted in fig. 3, where the third processor of the TMR node has a VMQ in a state different from the other two. Assume that all the processors are non-faulty. It is quite likely that the results produced by processes P₁ and P₂ for message m₂ are different from those produced by P₃. Processor P₃ can thus appear to behave like a faulty processor.
4. Task Scheduling Approaches

In a concurrent processing environment, the sequencing condition can be particularly hard to meet. As an example, consider an NMR node $N_k$ which can receive requests from two different NMR nodes $N_i$ and $N_j$ (fig. 4). Suppose $N_i$ and $N_j$ send their (replicated) results to $N_k$ at about the same time and that messages can experience variable transmission delays. It is thus possible for $N_i$ and $N_j$ messages to be voted at $N_k$ voters in a different order: this can cause - unless some preventive measures are taken - the messages to be queued in a different order in the VMQs of node $N_k$.

There can be several ways of meeting the sequencing condition, some of which are briefly discussed here.

(1) **Use of atomic broadcasts**: An atomic broadcast message sending facility exhibits the following three properties [6]: (i) it delivers every message broadcast by a non-faulty sender to all non-faulty receivers within some known time bound (*termination*); (ii) it ensures that every message whose broadcast is initiated by a non-faulty sender is either delivered to all correct receivers or to none of them (*atomicity*); and (iii) it guarantees that
all messages delivered from all non-faulty senders are delivered in the same order at all non-faulty receivers (order). It can be seen that by employing atomic broadcasts, it can be ensured that all non-faulty voters of a node receive messages in identical order, which in turn makes the task of meeting the sequencing condition straightforward. The remaining approaches do not rely on atomic broadcasts.

(2) **Use of an agreement protocol:** In this approach [7], non-faulty processors of an NMR node periodically take part in an agreement protocol to ensure that their respective VMQs are identical to each other, thereby guaranteeing that the sequencing condition is met.

(3) **Identical message selection:** This approach [5] requires that all of the non-faulty task schedulers of a node pick up messages from their voted message pools in an identical order for queueing in their respective VMQs.

(4) **Exception detection with recovery:** In contrast to the previous three approaches that prevent the occurrence of sequencing failures, this approach employs the philosophy of fault tolerance: sequencing failures are permitted, but are detected by voters as exceptions; specific exception handlers can then be employed for recovering from such failures [8].
As far as we know, there is little, if any, operational experience with distributed NMR systems, so a comparative evaluation of the various approaches discussed here does not appear to be possible. Note, however, that assuming that failed processors can exhibit arbitrary behaviour - the first two approaches require the use of Byzantine agreement protocols and therefore appear less attractive than the remaining two approaches. There is however a 'hidden' synchronization overhead in all of the approaches presented here; this is the requirement that the clocks of all of the non-faulty processors be kept synchronized to some accuracy. If we assume that a faulty clock can exhibit arbitrary behaviour, then any clock synchronization algorithm must rely on a Byzantine agreement protocol [see 9 for example]. We thus see that the need for agreement protocols can not be ruled out entirely in replicated systems with voting.

5. Failure Detection

The possibility of detecting sequencing failures as exceptions during voting was mentioned earlier. We can go a step further and use voting for detecting processor failures and node failures (a node will fail when a majority of its processors have failed). Detecting the latter two types of failure is extremely important in systems that require on-line reconfiguration. The details given here are, of necessity, rather sketchy; for a detailed exposition the reader is referred to a previous paper [8].

We require the facility of authenticating messages [10] before voters can be used for detecting the above mentioned failures. We assume that an actuator request is a triple \(<s(t_i), t_i, r_i>\), where \(r_i\) is the data part containing an encoding of the processing requirements, \(t_i\) is a unique timestamp, and \(s(t_i)\) is a unique signature derived from \(t_i\). During the processing of a request, only the data part will be modified (as the message travels from one node to the other). The objectives of message authentication are to ensure that any corruption of a
message or a forgery of the signature of a message can be detected by a non-faulty receiver. As a result, a non-faulty voter can maintain a pool of only authenticated messages for voting. Assuming that a voter votes as soon as a majority of messages from a given node are available, it can detect the following situations:

(i) All the messages being voted upon are identical. This represents the normal situation.

(ii) The timestamps of the messages being voted upon are not identical. This can only mean that the node supplying the messages has suffered a sequencing failure.

(iii) The timestamps of the messages being voted upon are identical, there is a disagreement in the data part of the messages, but a majority vote is possible. This can only mean that those processors supplying the disagreeing messages failed during the processing of the messages.

(iv) Same as (iii) except no majority vote is possible. This indicates a failure of the node supplying the messages during processing.

We thus see that, in addition to masking failures, a voter can also perform the important task of detecting them.

6. Concluding Remarks

Functionally distributed real-time systems are no longer a thing of the future - they are here and in regular use. For example, the on-board computing system for the F/A-18 aircraft contains fourteen computers (twelve for performing sensor oriented computations and two for performing mission oriented computations) connected by a bus [11]. This system employs a number of ad hoc techniques for achieving fault tolerance. Modular redundancy in the form of replication of processing modules with majority voting provides a systematic and powerful means of introducing fault tolerance in systems. We
have discussed an architecture for replicated processing of computations in
distributed NMR systems, highlighting the various task scheduling approaches
to meet the sequencing condition. In addition, the use of majority voters for
detecting failures in the system was also briefly discussed. We believe that the
concepts presented here form a sound basis for constructing highly reliable
distributed systems. We conclude this paper by observing that the architecture
presented here can be adapted to run diverse software (e.g. N-version programs
[12]) to obtain a measure of tolerance against design faults in application
programs.

References
[1] M. Pease, R. Shostak and L. Lamport, 'Reaching agreement in the presence
222-238.
processors' ACM TOCS, May 1984.
processing in distributed real time systems', Tech. Report, Computing
Laboratory, University of Newcastle upon Tyne (to appear).
simple message diffusion to Byzantine agreement', Digest of papers, FTCS-


