A Software Approach to the Construction of Fail-Controlled Nodes for Distributed Systems

Francisco Vilar Brasileiro

Departamento de Sistemas e Computação
Universidade Federal da Paraíba - UFPb/Campus II
Av. Aprígio Veloso s/n, Campina Grande - Pb, 58109-970, Brasil

Resumo
Sistemas distribuídos são normalmente projetados de forma a exercer um controle sobre as possíveis falhas que a infra-estrutura computacional e de comunicação do sistema venha a sofrer. Embora por um lado o tratamento efetivo de tais falhas possa garantir o provimento dos serviços oferecidos pelo sistema, por outro lado, é também o principal responsável pelo aumento na complexidade do projeto do sistema. Uma maneira de diminuir essa complexidade é assumir que o sistema irá executar sobre uma plataforma que falhe somente de uma forma controlada e previsível. Infelizmente, quando processadores convencionais são usados, não é sempre possível fazer esse tipo de consideração (mesmo quando os processadores são construídos com tecnologia `estado-da-arte'). Desta forma, faz-se necessário construir unidades de processamento, ou nodos, que, com uma probabilidade suficientemente alta, falhem somente da maneira assumida. Este artigo discute nossa experiência na construção de tais nodos. Os nodos apresentados neste artigo são formados exclusivamente por processadores convencionais, que são controlados por protocolos implementados em `software', e que não necessitam quaisquer circuitos especiais para o controle de sua redundância. Nodos com diferentes características foram implementados, e submetidos a uma variedade de experimentos, os quais indicam que os nodos implementados constituem uma solução prática para uma classe abrangente de aplicações.

Abstract
Designing and implementing distributed systems which continue to provide specified services in the presence of processing sites (or nodes) and communication failures is a difficult task. To facilitate their development, distributed systems have been built assuming that their underlying hardware components are fail-controlled, i.e. present a well defined failure mode. However, if conventional hardware cannot provide the assumed failure mode, there is a need to build nodes that present, with sufficiently high probability, the fail-controlled behaviour assumed. Our experience in constructing such nodes is discussed in this paper. The fail-controlled nodes described in this paper are built utilising `off-the-shelf' processors and software protocols to control the redundant components of the node, without recourse to any specialised hardware. We have implemented both failure-masking and fail-silent nodes, and have subjected them to a variety of experiments, which indicate the feasibility of using them in a wide range of applications.

Keywords and Phrases
Distributed processing, fault tolerance, replicated processing, fail-silence, failure-masking.
1. Introduction

Experiences in constructing distributed systems which continue to provide specified services in the presence of processing site and communication failures, have shown that designing and implementing such systems is a difficult task. In a perfect world, one would like to construct a distributed system using hardware components which are guaranteed to be either failure-free or to have well defined failure modes. However, all hardware components must fail eventually, possibly in an unpredictable manner [La86]. A sensible approach, taken by the designers of a considerable number of dependable distributed systems reported in the literature, is to build their systems assuming that the underlying hardware components are fail-controlled [Lap89], i.e. present a well defined failure mode, and then build processing sites or nodes and communication infra-structure that do indeed present the fail-controlled behaviour assumed.

A well studied way of constructing fail-controlled nodes is to couple redundant processors within a replicated node. A fail-controlled replicated node is composed of a number of processors which fail independently. Computation is replicated and executed simultaneously at each processor. By employing a suitable validation technique to the outputs generated by the replicated processors (e.g. majority voting, comparison), outputs from faulty processors can be prevented from appearing at the application level.

A number of ways of constructing replicated nodes have been reported in the literature (e.g. [Sm84, La86, Be88, WB91]). These nodes are based on hardwired mechanisms to couple replicated processors with specialised validation hardware circuits (e.g. comparator, voter), and therefore we name them hard nodes. In hard nodes, processors are tightly synchronised at the clock cycle level, and have their outputs validated at appropriate times by a reliable validation hardware. Another strategy, pioneered by the designers of the SIFT system [W*78], is to implement fail-controlled replicated nodes by using software mechanisms to perform both synchronisation of redundant processors and validation of the outputs of the node. We refer to these nodes as soft nodes.

The main advantages of hard nodes are the minimum performance overhead incurred, and the small impact that the architecture imposes on the software design process. However, there are also some problems with this approach. Firstly, individual processors must be built in such a way that they have a deterministic behaviour at each clock cycle. This can rule out the utilisation of `off-the-shelf' processors, whose reliability is normally higher than specially designed processors [SS92]. Secondly, the introduction of special circuits such as reliable comparator/voter and synchronisation mechanisms increases the complexity of the design, which at an extreme can result in a decrease in the overall node reliability. Finally, every new microprocessor architecture requires a considerable redesign overhead.

The advantages of soft nodes are very much the converse of the drawbacks discussed above. The absence of tight synchronism allows the utilisation of `off-the-shelf' processors. Further, by employing different types of processors within a node, there is a possibility that a measure of tolerance against design faults [Lap89] in processors can be obtained, without recourse to any specialised hardware assistance. Another advantage is that software protocols are much more flexible than their hardware counterparts. Also, the fact that the redundancy management protocols are implemented in software, allows the design of the underlying hardware to be made much simpler and possible to scale. Finally, technology upgrades appear to be easy, as the principles behind the protocols do not change, and software protocols can be ported relatively easily to any type of processor (including those expected to be available in the future). There is however, a major concern over the performance overhead incurred by the redundancy management protocols. In SIFT, for instance, the overhead associated with redundancy management can consume as much as 80% of the processor throughput [PB85].

Hybrid solutions, which incorporate both tight synchronisation and software synchronisation mechanisms, have been proposed to reduce this overhead. MAFT [KWFT88], FTP-AP [LA88] and Delta-4 [Po92] are hybrid architectures which share the same general structure. These architectures are structured around a tight synchronised hard core, on top of which conventional processors are replicated. The tight synchronised hard core is responsible for executing management functions, whilst
application processes are executed at the upper level replicated processors. The extra computational power delivered by the replicated processors increases the throughput of the system, and provides all the advantages of the software synchronisation approach; however, the underlying hard core re-introduces the problems associated with tight synchronisation.

We have investigated alternative ways of constructing efficient fail-controlled soft replicated nodes based solely on the utilisation of ‘off-the-shelf’ processors (which can fail in an arbitrary way, although restricted by authentication capabilities [SDC90]) and software protocols to control system redundancy, without recourse to any specialised hardware. Our approach to the construction of fail-controlled nodes is to follow the state machine model adopted by the Voltan family of soft replicated nodes [SESTT92, STBES93]. We have sought means of optimising the performance of those nodes by developing more efficient redundancy management protocols. In particular, we have developed much more efficient order protocols [BESST92, Br95a], which are necessary for the implementation of such nodes. In this paper we present the general structure of our soft fail-controlled nodes, followed by a brief description of the order protocols that we have used to implement them. We then describe the various nodes we have implemented and the experiments we have carried out to analyse their performance. The performance evaluation of the nodes implemented indicate the feasibility of utilising them in a wide range of applications.

2. The Voltan Architecture

A fail-controlled replicated node offers a service that can be characterised by its operational semantics and its failure semantics (failure mode). The operational semantics corresponds to the standard specification of the node’s service, whilst the failure semantics describes the behaviour of the node when up to a bounded number of components failures, which the node is able to tolerate, have occurred. Any behaviour that the node may present, which is not specified by either its operational semantics or its failure semantics, is considered to be exceptional behaviour. Further, the node presents an exceptional behaviour only if the number of components failures that the node experiences exceeds the number of components failures that the node is designed to tolerate, i.e. the bound specified on its failure semantics. An upper level system can then assess the suitability of using the services of a particular underlying fail-controlled replicated node by analysing the operational and failure semantics of the latter, and by estimating the likelihood of the occurrence of exceptional behaviour. Alternatively, one can build a fail-controlled replicated node whose probability of presenting exceptional behaviour is sufficiently small, so that the node delivers, with the necessary probability, its particular fail-controlled behaviour.

The Voltan family of fail-controlled soft replicated nodes includes failure-masking and fail-silent nodes [SESTT92]. Failure-masking nodes possess failure semantics which is equivalent to their operational semantics, that is, the node still delivers its standard service despite the occurrence of a bounded number of components failures, which are masked, whilst fail-silent nodes possess a safe [Lap89] failure semantics, that is, after the detection of the failure of any node component, the node neither delivers its standard service, nor unspecified ones, and simply halts.

2.1. System Model

We assume that (non-replicated) distributed applications are composed of a number of processes that do not share memory, and interact only via messages. As an example, the function of a typical ‘server’ process is to cycle by selecting an input message from any one of its input ports, process it and, if necessary, output one or more messages on its output ports. We also assume that if a process with multiple input ports has input messages available at several of these ports, then any one of these messages is chosen non-deterministically for processing. Message selection is however assumed to be fair, that is, the process eventually selects a message present on a port.

In the replicated version of a process executing on a replicated node, multiple input ports of the non-replicated process are merged into a single port and the replica selects the message at the head of its port queue for processing. Hence, provided the queues of all correct replicas can be guaranteed to contain identical messages in an identical order, and all the non-faulty replicas have identical initial states, then
identical output messages are produced by them [Se90]. Validation protocols can then be applied to the outputs generated by replica processes. Thus, replication of a process requires the following two conditions to be met:

**agreement**: all non-faulty replicas of a process receive identical input messages; and

**order**: all non-faulty replicas process the messages in an identical order.

Note that output messages are identical only if the computation performed by a process on a selected input message is deterministic. Practical distributed programs often require some additional features such as using time-outs when waiting for messages. Time-outs and other asynchronous events, high priority messages, etc., are potential sources of non-determinism during input message selection, making such programs difficult to replicate. In [TS90], Voltan nodes are enhanced with the necessary functionality for dealing with such cases. In this paper, we assume the simple state machine model discussed above, where processes are assumed to be deterministic. So, if processes present this deterministic behaviour, active replication is achieved by providing suitable protocols for achieving agreement and order of input messages, and for validating output messages.

### 2.2. General Assumptions and Node Architecture

We consider failure-masking nodes and fail-silent nodes comprised of $N$ processors, where $N = 2\xi + 1$ in the case of failure-masking nodes, $N = \xi + 1$ in the case of fail-silent nodes, and $\xi, \xi > 0$, is the upper bound on the number of processors of a node that may fail. Each node in the system, each processor of a node, and each group of replicated application processes possess unique identifiers (numbers). There is also a sequence number counter associated with each group of replicated application processes. These identifiers and counters are used to uniquely identify each message generated by any application process. Messages generated by an application process are encapsulated with control information which contains the message's sequence number, process identifier, node identifier, and authentication information. This information is used by the redundancy management protocols to match correlated messages, and to detect and remove duplicated and corrupted messages.

We assume that mechanisms exist for generating and verifying digital signatures, which provides an authentication facility with arbitrary high probability [RSA78]. Each non-faulty processor has a mechanism which generates a unique, message dependent, unforgeable signature, which is attached to any message it sends to other processors of the node. Furthermore, every non-faulty processor is also assumed to have an authentication function for verifying the authenticity of a message signature contained in a received message. We assume that a faulty processor (and therefore the processes running on that processor) can fail in an authentication-detectable arbitrary way [SDC90], where the class of authenticated-detectable arbitrary failures is defined as the class of arbitrary failures that do not corrupt the authentication mechanisms described above.

Each processor of a node is assumed to have network interfaces for inter-node communication over (possibly redundant) networks. In addition, the processors of a node are internally connected by communication links for intra-node communication needed for the execution of the redundancy management protocols. No bound is assumed on network transmission delays between distinct nodes. We assume that the maximum intra-node communication delay over a link is bounded by a known constant $\lambda$, i.e. a message whose broadcast to a number of receivers within a node is initiated by a non-faulty sender, experiences an actual transmission delay of $\lambda$ units of real time, $\lambda < \lambda$ until it is received by every non-faulty receiver. For simplicity, we assume that the lower bound on the actual transmission delay is zero. Thus, $0 < \lambda$, and $\lambda$ also represents the maximum variation in message transmission delays within a node. Link failures are categorised as processor failures, that is, a link failure that prevents a message sent from a processor to be received by other processors of the node is considered as a failure of the sender processor. A processor measures the passage of real time via its physical clock. We further assume that the clock of a non-faulty processor can drift from real time by a bounded and known rate $\rho$, $|\rho| < \rho$. Note that these assumptions make the environment synchronous, and are essential to guarantee that the agreement necessary for ordering input messages is reached in finite time in the presence of failures [FLP85], and thus to ensure the liveness of processing activities within the replicated node.
The general structure of a Voltan node is as follows. In addition to application processes (Server processes), each non-faulty processor of a node executes five system processes, namely Sender, Validator, Transmitter, Receiver and Order processes. Communication between two processes executing at the same processor is realised through message queues and message lists data structures. Figure 1 shows the inter-communication paths between the processes executing at any non-faulty processor of a Voltan node.

Figure 1: processes executing at a non-faulty processor of a Voltan node

The function of each system process is described below.

Sender process: this process takes the messages produced by the application processes of that processor, signs them and sends them to the other processors of the node for validation, i.e. voting in failure-masking nodes, and comparison in fail-silent nodes.

Validator process: the function of this process depends on the type of the node. In failure-masking nodes, the Validator processor is a Voter process. It compares authentic messages which have been signed and sent by other processors with their counterparts produced locally. If the comparison is not successful, the message is discarded. Otherwise, the message is countersigned (by considering the existing signatures in the message as part of the message), and if there are now $\ell+1$ signatures in the message, the message, termed a valid message, is handed over to the local Transmitter process for network delivery to destination nodes. If there are less than $\ell+1$ signatures, then the message is sent to the other processors of the node that have not signed the message yet. In fail-silent nodes the Validator process is a Comparator process. It also compares authentic messages which have been signed and sent by other processors with their counterparts produced locally. If the comparison is successful, the message is countersigned, and if there are $\ell+1$ signatures in the message, the valid message is handed over to the local Transmitter process. If there are less than $\ell+1$ signatures, then the message is sent to the other processors of the node that have not signed the message yet. A comparison that detects a disagreement indicates a failure. Similarly, an absence of a $\ell$-signed message for comparison (after a node specific time-out interval) also indicates a failure. Once a failure is detected, the Comparator process terminates its activities, and so does the Sender process. Hence, no new $\ell+1$-signed messages are ever output by the node. It follows that, in both failure-masking and fail-silent nodes, all valid messages issuing from a node contain $\ell+1$ signatures.

Transmitter process: this process is responsible for sending the $\ell+1$-signed messages over the network to destination nodes.

Receiver process: this process authenticates messages received from the network or from the internal links and discards any message which fails authentication or any duplicated message received. Authenticated messages from the network (valid messages) are sent to the local Order process. Authenticated messages received from other processors of the node, which carry less than $\ell+1$ signatures, are sent to the local Validator process.

Order process: this process executes an order protocol with its counterparts in the other processors of the node. The function of the order protocol is to construct identical queues of valid messages received from the network for processing by the local application processes of all non-faulty processors of the node. (Since order protocols entail the Order process to relay valid messages to its counterparts, it is sufficient for a message to be received from the network by any one of the non-faulty processors of a node for it to be ordered at all the non-faulty processors.)

As mentioned before, processes executing at the same processor communicate via queue and list data structures. The basic difference between a queue and a list is that a process is only allowed to access messages at the head of the queue, whilst processes can access any message that has been deposited in a list. The following queues and lists are used:

- **Received Message Queue (RMQ):** Contains valid messages intended for ordering, that have been received from the network and authenticated by the Receiver process.
- **Delivered Message Queue** (DMQ): Contains ordered messages to be consumed by the application process Server.
**Processed Message Queue (PMQ):** Contains unsigned output messages produced by local application processes. These messages must be validated by the Validator process before transmission to the final destination.

**External Candidate Message List (ECL):** Contains authenticated signed messages that have been received from other processors for validation.

**Internal Candidate Message List (ICL):** Contains unsigned messages, each waiting for matching signed messages to arrive in ECL.

**Validated Message Queue (VMQ):** Contains \( \mathbb{P} + 1 \)-signed, valid messages ready to be transmitted over the network.

The Receiver, Sender, Transmitter and Validator processes described above are of trivial implementation. Thus, we discuss their implementation in this section, whilst we leave the discussion of the implementation of the more complex Order process to the next section.

The Receiver process is composed of two cyclic processes that execute in parallel. The first process receives messages from the network, and performs an authentication check on the messages' signature, which detects any spurious message that has been sent by the faulty processors of other replicated nodes. Authentic messages are deposit into the local RMQ. The second process receives messages from the internal links. These messages are also subjected to authentication, and if found to be authentic are deposit into the local ECL. Authentication in this case detects internal errors, i.e. any attempt of a faulty processor of the node to corrupt a message sent through an internal link. The Sender process cycles removing messages from the local PMQ, depositing them into the local ICL, and diffusing them to the other processors of the node that have not received that message yet (i.e. those whose signature are not present in the message). The Transmitter process cycles removing messages from the local VMQ and transmitting them to the appropriate destination nodes. The Validator process has access to the ICL, which contains messages produced locally by the application processes, and to the ECL, which contains authenticated signed messages with up to \( \mathbb{P} \) signatures, which have been produced by the application processes executing at the other processors of the node. Messages in ECL may have been subjected to evaluation at other processors of the node; essentially, a message which contains \( s \) signatures has been successfully compared against the locally produced messages of \( s - 1 \) other processors. The function of the Validator process is to scan ECL looking for signed counterparts of the local messages present in ICL. When such a pair is found, then the appropriate validation action is taken. The validation action is what differentiates the Voter process of a failure-masking node from the Comparator process of a fail-silent node. The Voter process compares the contents of the messages, and if there is a mismatch, the message in ECL is discarded. Otherwise, the message in ECL is countersigned and either enqueued in the local VMQ for later transmission to the destination by the Transmitter process, or sent to all other processors of the node that have not signed the message yet. Messages signed by a majority of processors, i.e. \( \mathbb{P} + 1 \)-signed messages, are valid messages, and thus are enqueued in VMQ. Messages with less than \( \mathbb{P} + 1 \) signatures are sent through the internal links to the other processors of the node. The functioning of the Comparator process is similar to that of the Voter process, except that once a mismatch is detected in any message comparison, the Comparator process must bring the node to a halt. Upon the detection of a failure, a non-faulty processor can halt the node by firstly stopping the activities of its Comparator process, so that no new \( \mathbb{P} + 1 \)-signed messages are ever output by that processor, and secondly stopping the activities of its Sender process, so that the other processors of the node do not output new \( \mathbb{P} + 1 \)-signed messages too.

The main difference between a distributed system composed of Voltan nodes, and another composed of non-replicated nodes, is that in the former, nodes are required to produce \( \mathbb{P} + 1 \)-signed messages and use authentication to distinguish between valid and spurious messages. Further, under normal conditions, inter-node traffic is increased \( 2 \mathbb{P} + 1 \) times when the distributed system is composed of failure-masking nodes, and \( \mathbb{P} + 1 \) times when the nodes are fail-silent.

Provided that the number of faulty processors does not exceed \( \mathbb{P} \), the semantics of the failure-masking and fail-silent nodes presented above can be expressed in the following way:
**failure-masking nodes**: when all processors of a failure-masking node are non-faulty, the node produces 2¶+1 copies of valid output messages, that can be verified as such by any non-faulty processor of the destination nodes. When up to ¶ processors are faulty, a failure-masking node outputs at least ¶+1 copies of valid output messages, and may also output spurious messages. Any spurious message output by the faulty processors of a failure-masking node can be detected and rejected by all non-faulty processors of any receiver node;

**fail-silent nodes**: when all processors of a fail-silent node are non-faulty, the node produces ¶+1 copies of valid output messages, that can be verified as such by any non-faulty processor of the destination nodes. When any number of processors, up to ¶, are detected as faulty by any non-faulty processor within the node, the node ceases to produce new valid output messages, in which case non-faulty processors of receiver nodes can detect any message it may produce as unwanted.

In both nodes, spurious messages are detected by examining the signatures attached to the messages received, whilst the sequencing information of messages is used to detect duplicates.

### 3. The Order Process

The function of the Order process is to ensure that authentic valid messages received by the non-faulty processors of the node, i.e. messages enqueued in RMQ, are enqueued in the same order in the appropriate DMQi of every non-faulty processor of the node (see Figure 1). Ordering can be achieved in several ways. The basic idea is to implement an agreement protocol which guarantees that all non-faulty replicas receive the same set of messages and then accomplish ordering by assigning monotonically increasing sequence numbers to messages. Agreement is normally achieved via message diffusion techniques. It is also necessary to devise a method to establish when a particular message has become stable, i.e. define the instance of time when it is guaranteed that no timely messages with sequence numbers less than a certain value, say \( sn \), will ever be received, therefore all messages with sequence numbers less than \( sn \) can be processed in a consistent order among all non-faulty replicas.

#### 3.1. Using Synchronised Time to Achieve Order

In a synchronous system, atomic broadcast protocols (e.g. [CASD85]) can be used to achieve ordering. It suffices that every message in the RMQ of a non-faulty processor is atomically broadcast to all other processors of the node. Most atomic broadcast protocols require clocks of all non-faulty processors to be synchronised such that the measurable difference between the readings of any two clocks at any instance of real time is bounded by a known constant, say \( \epsilon \). The ordering of input messages using an atomic broadcast protocol works as follows. The Order process of a processor time-stamps a message to be ordered with its local clock reading. A copy of the time-stamped message is signed and sent to the Order process of the other processors of the node. The Order process implements a message diffusion mechanism which guarantees agreement on the messages received. The message diffusion mechanism incorporates timeliness checks that allow the Order process to discard any untimely message received. If \( t \) is the time-stamp of a message received from or sent to the Order process of the other processors, then this message becomes stable at local clock time \( t+\Delta \), where \( \Delta \), the termination bound of the atomic broadcast protocol, is given by: \( \Delta = (\|+1)(d_\Delta+\epsilon) \), where \( d_\Delta \), \( d_\Delta = \_/(1-\rho) \), is the minimum clock time interval that the clock of any non-faulty processor must advance to measure the real time interval \_. Stable messages are enqueued in the appropriate DMQi in increasing time-stamp order, with the action being taken to discard, rather than to enqueue a stable message, if its replica has already been enqueued. (The identifier of the processor which has initiated the broadcast of a particular message is used as a tie-breaker in the case when there are more than one stable messages with the same time-stamp.)

The Order process incorporating the protocol discussed above is composed of three cyclic processes, namely Broadcast, Diffuse and Deliver processes, which execute in parallel. This three processors communicate with each other through a shared message list called the Ordered Message List (OML). The Broadcast process picks up a message from the RMQ, time-stamps it with its current clock reading, signs it and then sends it to the other processors of the node. It also inserts a copy of the message into
the OML. The Diffuse process receives diffused messages from the other processors. After checking the authenticity of a message received from another processor, the Diffuse process performs a timeliness check that allows it to discard any message received too early (messages with time-stamp greater than $c + (s\Delta)$, where $c$ is the current reading of the processor's clock, and $s$, $1 \leq s \leq \frac{\rho}{\Delta} + 1$, is the number of signatures contained in the message, i.e. the number of processors that have diffused the message), or any message received too late, i.e. messages with time-stamp less than $c - s\Delta + \epsilon$.

Authentic and timely messages are accepted and inserted into the OML. Corrupted and untimely messages are discarded. The Diffuse process also checks if the message received is a copy of a previously received message, in which case the message is also discarded. If an accepted message has been signed by less than $\frac{\rho}{\Delta} + 1$ processors, then the Diffuse process signs and diffuses the message to the other processors in the node which have not received that message yet.

The Deliver process takes stable messages (messages with time-stamp less than $c - \Delta$) from the OML, filters duplicate messages (i.e. removes all the duplicated copies of a particular message, but one), removes spurious messages (messages diffused by the same source, with the same time-stamps, but with different contents) and enqueues the remaining messages in the appropriate DMQs in increasing order of time-stamps.

The termination bound ($\Delta$) of the atomic broadcast protocol gives the delay suffered by an input message before it is made available to the application process of a particular processor, as measured by the clock of that processor. However, the node can only output the results of the computation of a particular input message when all non-faulty processors have ordered and processed the input message. Hence, we define the actual stability delay ($\Delta_a$) of an order protocol for a particular message to be the real time elapsed since a copy of the message is first received by a non-faulty processor of the node until it is ordered and enqueued in the appropriate DMQs of all non-faulty processors of the node. Therefore, for the above protocol we have:

$$\Delta(1-p) \leq \Delta_a \leq \Delta(1+p)+\epsilon.$$

The lower bound is achieved when the actual difference between the readings of the clocks of any two non-faulty processors at the time the message is being ordered is zero, and the clocks of all non-faulty processors are running at the fastest possible rate. On the other hand, the upper bound is achieved when the clock of at least one non-faulty processor is running at the slowest possible rate, and the difference between the reading of the clock of the first non-faulty processor to receive a copy of the message, and the reading of the slowest non-faulty processor at the time the message is being ordered is equal to $\epsilon$.

### 3.2. Using Time-Outs to Achieve Order

As discussed before, synchronous and deterministic atomic broadcast protocols published in the literature require that non-faulty processors have access to synchronised clocks. Since physical clocks of non-faulty processors drift from real time by different rates, meeting this requirement demands each processor to periodically compute the amount of adjustment to be made to the reading of its local physical clock, and then to make the adjustment so that the synchronous property of the clocking system is maintained. However, if the atomic broadcast protocol is executed frequently enough, adjustments can be computed with no message overhead [BD87].

In a Voltan node, a given input message can prompt more than one processor to initiate an execution of the atomic broadcast protocol. By exploring the execution of these protocols, it is possible to attain some sort of synchronisation among processors, without the need for executing an explicit clock synchronisation protocol. This observation has motivated us to develop atomic broadcast protocols that do not require the maintenance of the synchronised-clock abstraction, but rely directly on physical clocks for knowing the current time and for scheduling operations at future times [Br95b].

In [Br95a] a time-out based order protocol based on the atomic broadcast protocol of [Br95b] is presented. Its performance is comparable with that of the synchronised-time based protocols reported in the literature, but, unlike the latter, does not require the maintenance of explicitly synchronised clocks. In that protocol, each non-faulty processor maintains a message counter which is used to time-stamp messages it broadcasts, and timing counters which are used to determine the timeliness of diffused messages received from the other processors of the node. There is a timing counter associated with each
number $s$ of processors ($1 \leq s \leq \lceil \rho+1 \rceil$) that may have diffused a message. A message diffused by $s$ processors before being received by a non-faulty processor and which bares a time-stamp not larger than the value of the correspondent timing counter is considered to be untimely and is discarded. The value of the message counter is incremented after each broadcast, and it is updated in such a way that its value is always larger than the value of any of the timing counters. On the other hand, timing counters are update so to conform with the following rule:

**timeliness criterion:** a received message with time-stamp $ts$ which has been diffused by $s$ processors is timely only if it is received before local clock time $t+2sd$, where $d = \rho/(1-(2\rho+1)\rho)$, and $t$ is the local clock time when the value of the message counter of the receiver processor first became larger than $ts$ (see [Br95a]).

The message counter is updated either when the processor initiates the broadcast of a message or when it receives a timely message with time-stamp larger than the current value of its message counter, thus, the stability delay for a particular message is $2d(\rho+1)(1+\rho_d)$ for the processor that broadcasts the message, and $2d(\rho+1)(1+\rho_d)+\_d$ for the other processors, where $\rho_d$ is actual rate with which the processor's clock drift from real time and $\_d$ is the actual transmission delay for a message received from the network to be relayed to the other non-faulty processors of the node. Hence, the actual stability delay of the protocol is given by:

$$2d(\rho+1)(1-\rho) + \_d + 2d(\rho+1)(1+\rho) + \_d.$$ 

In practice, three-processor failure-masking nodes (TMR nodes) are the most commonly selected replication strategy for masking nodes. [Br95a] also presents an improved version of the above protocol for TMR nodes. The protocol exploits the fact that each processor has only two other processors to reach agreement with, and achieves a performance that is better than any synchronised-time based protocol published to date. For a TMR node, the stability delay of the modified protocol for a particular message is $4d(1+\rho_d)$ for the processor that broadcasts the message, and $3d(1+\rho_d)+\_d$ (instead of $4d(1+\rho_d)+\_d$, as per the previous protocol) for the other two processors. The actual stability delay of a message for a TMR node using this improved order protocol is given by:

$$4d(1-\rho) + \_d + 3d(1+\rho_d).$$

It is worth noting that within the architecture of our failure-masking nodes, it is possible that every non-faulty processor of a node receives a copy of a particular message from the network. Therefore, in a TMR node, copies of the message that have been relayed by other processors of the node may become stable at all non-faulty processors prior to the time when the message they have received from the network is due to stabilise. If we assume that $\lambda_d, \lambda_a = 0$, is the actual message reception skew for a particular message, i.e. the difference between the real times when the first two copies of a particular message are received from the network by any two non-faulty processors of the node, then a relayed message will become stable at any non-faulty processor of the node at latest by:

$$\lambda_a + \_d + 3d(1+\rho_d).$$

Thus, whenever $\lambda_a + \_d$ is smaller than $d(1-\rho)$, the lower bound on the actual stability delay of a message is given by $\lambda_a + \_d + 3d(1-\rho)$. Note that if only one non-faulty processor receives a particular message from the network and the other non-faulty processors do not receive a copy of the same message from the network, then $\lambda_a = \_d$, but the message is still ordered at all non-faulty processors of the node.

The above protocol can be enhanced with an acknowledgment mechanism so that in the absence of failures, the protocol’s actual stability delay can be dramatically reduced. [Br95a] shows how to modify the above time-out based protocol to implement an *early-order* protocol, so that, whenever the node is in a failure-free state, it achieves much better performance. In the modified protocol, whenever a non-faulty processor receives a timely message from another processor, bearing a time-stamp larger than the time-stamp of the latest message it has broadcast, then a null message is broadcast. Null messages have the sole purpose of speeding up the stabilisation of messages, and therefore are discarded before delivery. The upper bound on the stability delay of the early-order protocol is the same of that for the original time-out based protocol. On the other hand, in the absence of failures, the protocol’s maximum actual stability delay is given by $(2+\rho)\_d$ (see [Br95a]).

### 3.3 Order Protocols for Fail-Silent Nodes
The order protocols discussed so far can be used to implement the Order process of both failure-
masking and fail-silent nodes. However, by exploiting the particular characteristics of fail-silent nodes, it
is possible to design a much simpler and more efficient Order process. The Comparator process of a
fail-silent node must bring the node to a halt as soon as a failure is detected, thus, in a fail-silent node,
the task of coping with any failure in achieving a correct order of input messages can be delegated to the
Comparator process. Therefore, unlike order protocols for failure-masking nodes, order protocols for
fail-silent nodes need not be fault-tolerant.

Apart from a simplified synchronised-time based order protocol, we have implemented two other
order protocols for the specific case of a two-processor fail-silent node. (The protocols can be easily
adapted for generic fail-silent nodes - see [Br92].) The first of them very much resembles the time-out
based protocol previously described. Each non-faulty processor maintains a message counter and a
single timing counter. The timing counter is updated whenever a timely message with time-stamp larger
than the timing counter’s current value is received from the other processor of the node. The message
counter is incremented after each broadcast, and it is updated in such a way that its value is always
larger than the timing counter’s value. Messages with time-stamp not larger than the value of the timing
counter are considered to be untimely and are discarded. In this way, a diffused message becomes stable
at the receiver processor as soon as it is received. There is also a time-out mechanism which is used to
stabilise messages at the sender processor. Basically, a message becomes stable at latest after a $2_\_d$ real
time interval has elapsed since it was sent to the other processor of the node. Since the two processors
act both as receivers as well as senders, the protocol’s actual stability delay is given by:

$$-a = \min \{ 2d(1+p_a), \lambda_a + -a \}.$$  

This time-out based order protocol has an actual stability delay that is affected by the message
reception skew $\lambda_a$. When $\lambda_a$ is large, the actual stability delay of the protocol deteriorates to its worst
case $2d(1+p)$. Thus, for systems where the message reception skew is large, this protocol does not solve
the problem of reducing the actual stability delay when ordering messages. We have developed an
asymmetric order protocol whose actual stability delay is not a function of $\lambda_a$, but rather a function of
$-a$ (see [Br92]). Thus, since our node model makes no assumptions on the upper bound of $\lambda_a$, this
protocol is more suitable for implementing the Order process of the nodes we are constructing.

In this asymmetric protocol we assign different roles to each of the two processors forming a node.
We term one processor the leader and the other the follower. It is the responsibility of the leader to
determine the order of messages for processing. Having selected a message from RMQ, the Order
process executing at the leader sends a copy of the message to the Order process executing at the
follower, which then processes messages in the order dictate by the leader (see [Br92]). The asymmetry
introduced by assigning different roles to the two processors of a node requires the introduction of an
extra mechanism in the follower for detecting late or non arrival of a message for ordering from the
leader. The Order process at the follower deposits each valid input message received from the network
in a local list with an associated time-out $t_r$, $t_r \leq 0$. Copies of authentic messages received from the
leader are compared against the messages in the local list. If a counterpart is found, then its time-out is
reset and both messages are discarded. If a time-out expires, the Order process at the follower assumes
that the leader has failed to send a message for ordering. This can happen either because the leader has
failed, or because the leader has not received a copy of the message from the network. The follower can
try to prevent a premature shut down (when the leader has not failed) by feeding the leader with the
missing input message. This `feedback' mechanism works as follows: after the time-out $t_r$ has expired,
the follower sends a copy of the missing input message to the leader in order to have it properly ordered.

Unlike the previous protocols, to calculate the actual stability delay of the asymmetric order protocol
it is relevant to identify the processor that first receives a copy of a particular input message. We define
$\lambda_{LF}$ as the message reception skew as perceived by the follower processor, i.e. the difference between
the real time that the leader receives a copy of a particular message from the network and the real time
that the follower receives a copy of the same message. The actual stability delay for this protocol is then
given by:

$$-a = _F = _L + -a; \text{ and } _L = \text{Erro!}$$
where \( F \) is the local stability delay for the follower, \( L \) is the local stability delay for the leader and \( \rho_f \) is the rate of the drift of the follower's clock.

A sensible strategy is for the follower to set \( t_f = 0 \) (thus, as soon as the follower receives a message from the network it checks for the presence of the corresponding diffused message from the leader). Hence, we have:

\[ a = F = L + a, \text{ and } L = \text{Erro!} \]

4. Implementation and Performance Evaluation

Efficient implementations of soft replicated nodes require that the processors forming a node be capable of exchanging messages quickly. Transputers are processing units with interfaces to fast point-to-point communication links, providing just the kind of functionality we require. For this reason, we have chosen to implement TMR and two-processor fail-silent nodes using a network of T800 Inmos transputers [INMOS88]. Each transputer has four internal links which connects it with four other transputers, providing a fast communication link. We have chosen to implement the protocols in an object oriented language, C++ [St92], and have used the facilities of the Helios operating system [Perihelion91] - a Unix-like operating system which runs on transputers and supports the client-server model for structuring programs. These choices are not central to our design and implementation, and have been taken mainly because we have extensive Unix/C++ based system programming experience.

Our implementation makes extensive use of classes, inheritance and virtual operations to implement the software architectures shown in Figure 1. Base classes exist for implementing processes, messages, queues and lists. The functionality of the system is then implemented in classes derived from these base classes.

4.1. Nodes Description

We have implemented three different versions of both TMR and two-processor fail-silent nodes. The nodes implemented are differentiated from each other only in the way input messages are ordered. We have used the synchronised-time and the time-out based order protocols discussed in section 3.1 and 3.2, respectively, to implement both TMR and fail-silent nodes. Further, we have implemented a TMR node incorporating the early-order, time-out based protocol of section 3.2 and a fail-silent node incorporating the asymmetric order protocol studied in section 3.3. We have also implemented a non-replicated node model which executes on a single transputer. Apart from the application processes, the non-replicated node incorporates Receiver and Transmitter system processes. (The Receiver process for the non-replicated node is slightly simpler than the one for replicated nodes, since input messages do not need to be authenticated.)

4.2. Experiments Description and Evaluation

The general structure of the experiments is that of a client application process executing on a non-replicated node, and which makes requests to a server application process executing on a replicated node. We are particularly interested in the overheads associated with ordering of input messages and validation of output messages. Thus, we have measured the following time intervals for the replicated nodes:

**Input delay (ID):** for the fail-silent nodes, the input delay measures the time interval between a message entering the node (the earliest time that a particular message is received by the Receiver process of any processor of the node) and the message being removed from the DMQ of both processors of the node; for the TMR nodes, the input delay measures the time interval between a message entering the node and the message being removed from the DMQ of a majority of the processors of the node (see Figure 1). Thus, the input delay is made up of the actual stability delay for a message \( a \) plus the time taken up by authentication and queue manipulation within the node; it reflects the overhead involved in ordering messages at a node.

**Output delay (OD):** for the fail-silent nodes, the output delay measures the time interval between a message becoming ready for comparison at both processors of the node (the latest time that the Server process of any processor of the node outputs its message) and the message being output by the node (the earliest time that a message is output by the Transmitter processor of any
processor of the node); for the TMR nodes, the output delay measures the time interval between a message becoming ready for voting at a majority of processors, and the message being output by the node (see Figure 1). The output delay reflects the time taken for a message to be validated and output.

**Node delay (ND):** the node delay is simply the sum of the input and output delays (ID+OD). It reflects the earliest response from a node to a given input message, i.e. the overhead associated with replication.

For the non-replicated node we have measured the following time interval:

**Response latency (RL):** the response latency is the time that the client will have to wait for the response for a particular request to arrive. It is made up by the small processing overhead associated with the output of a request and the reception of the correspondent response at the non-replicated node, the inter-node transmission delay for both the request and the response messages, and the processing overhead at the replicated node where the server executes.

### Overhead for a simple client-server application

In the first experiment a single client application process executing on a non-replicated node requests a simple service from the server application process which executes on a replicated node. The client process issues a request to the server process by broadcasting the request to the server process replicas executing on each of the processors forming the replicated node. It then waits for a response from any of the server replicas. Each server process replica executing on the replicated node receives a request from the client, services it (the actual computation performed is minimal) and sends the response back to the client. The client issues a new request upon reception of the first response message received from the server.

We have collected data for ten runs of the experiment, each run involving the client node sending 100 request messages of 64 bytes, and receiving response messages of the same size. For each run we have measured the input, output and node delays for the replicated node, as well as the response latency for the non-replicated node. For each one of these time intervals we have averaged the values measured for each of the requests processed. We have also measured the average message reception skew ($\lambda_{\text{av}}$) and the average link transmission delay of internal messages, making a distinction between the transmission delay of internal messages associated with the ordering of input messages ($\_\text{input}$) and the transmission delay of internal messages associated with the validation of output messages ($\_\text{output}$).

We have also executed the experiment for a server process executing on a single processor, i.e. on a non-replicated node. As we would anticipate, for the case of ordinary processors, the overheads are small. They exist because it is still necessary to enqueue and dequeue messages in the system. The measured node delay for the non-replicated node amounted to about 1.49 ms, of which about 0.75 ms was due to input overheads, whilst about 0.74 ms was due to output overheads. The average response latency measured was 5.74 ms.

We first concentrate the analysis on the overheads for the replicated nodes, starting with the overheads for the fail-silent node implementations. Table 1 summarises the average delays obtained for each fail-silent node implementation exercised.

<table>
<thead>
<tr>
<th>Model</th>
<th>ID</th>
<th>OD</th>
<th>ND</th>
<th>_input</th>
<th>_output</th>
<th>$\lambda_{\text{av}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail-silent synchronised-time based</td>
<td>21.59</td>
<td>6.31</td>
<td>27.90</td>
<td>2.70</td>
<td>1.46</td>
<td>0.41</td>
</tr>
<tr>
<td>fail-silent time-out based</td>
<td>8.45</td>
<td>6.67</td>
<td>15.12</td>
<td>2.77</td>
<td>1.40</td>
<td>0.40</td>
</tr>
<tr>
<td>fail-silent asymmetric</td>
<td>4.79</td>
<td>3.32</td>
<td>8.11</td>
<td>3.92</td>
<td>1.30</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table 1: Performance overhead for a client-server application on fail-silent nodes

For the case of the fail-silent node with synchronised-time based order protocol, experiments under worst case circumstances determined the smallest safe value for $\_\text{input}$ to be 12 ms. This implementation uses a simplified version of the clock synchronisation algorithm presented in [HSSD84]. As discussed in the previous section, we can assume a failure-free environment for the execution of the clock.
synchronisation protocol, thus allowing $\varepsilon$ to be set to $\frac{\lambda}{2}$. Hence we have fixed $\varepsilon = 6$ ms, which gives the stability delay, $\Delta$, of 18 ms (since $\Delta = \frac{\lambda}{2} + \varepsilon$). Measurements indicated that throughout the relatively small duration of the experiment, the actual synchronisation error was very small, thus, on average, the stability delay ($-\lambda$) is almost the same as $\Delta$, and the values shown in Table 1 for ID indicate that for this implementation the overheads due to message authentication and queue manipulation take up to 3.59 ms.

From the discussion in section 3, on average, the stability delay of the fail-silent node using the timeout based order protocol would be equal to the minimum value between $\frac{\lambda}{2}$ and $\frac{\lambda}{2} + \varepsilon$, plus any extra overheads. In our experiment $\frac{\lambda}{2} + \varepsilon$ is smaller than $\frac{\lambda}{2}$, thus, from the figures given in Table 1 the overheads due to message authentication and queue manipulation for this implementation take up to 5.28 ms. The increased overhead of this implementation suggests that in terms of execution time, the maintenance of timing counters of the time-out based implementation is more costly than the maintenance of synchronised physical clocks.

For the fail-silent node using the asymmetric order protocol, it is necessary to examine separately the performance of leader and follower processors since they are executing different protocols. From the analysis presented in the section 3, ID corresponds to the follower's stability delay ($-\lambda = -\lambda_F = -\lambda_L + \varepsilon$), plus any overheads due to message authentication and queue manipulation. In our experiment, the leader processor nearly always is the first processor of the node to receive a copy of a particular input message. Thus, most of the time we will have $\lambda_L < 0$, and consequently $-\lambda = -\varepsilon$. The values shown in Table 1 indicate that the overheads (0.87 ms) for the asymmetric implementation of a fail-silent node are close to those experienced by the non-replicated node. This is because the simplicity of the order protocol allowed its implementation to be incorporated into the Receiver process, eliminating the need for an Order process, and consequently reducing the overheads associated with context switching and queue manipulation. The overheads are slightly larger because in the replicated node messages must be authenticated. (Currently, simple checksums are being used as signatures to provide the authenticity function that our node model assumes, and so, have a relatively small impact upon system performance. The performance overhead of more complex signature mechanisms has not yet been assessed.)

Indeed, for small messages, most of the extra input overheads incurred by the different implementations of fail-silent nodes are due to context switching of processes and queue manipulation. This also contributes to the smaller overheads of the synchronised-time based implementation when compared to those for the time-out based implementation. In the former, the tighter synchronisation guaranteed by the time triggered characteristics of the synchronised-time based order protocol allows more efficient scheduling of the actions taken at each processor forming the node. We believe that a better control of the scheduling policy of the processors forming a fail-silent node can lead to a certain degree of improvement on the implementations of such nodes, particularly for the case of the time-out based implementation.

The positive effects of a more efficient scheduling strategy can also be noticed at the validation of output messages. Despite the fact that all fail-silent nodes implemented make use of the same comparison protocol, figures in Table 1 show that a node implemented with the asymmetric order protocol for input messages suffers less output delay than a node with a symmetric one. The reason for this is that the asymmetry introduced for input ordering helps the follower at comparison time: by the time a message becomes available in the follower's ICL, the leader's message will usually be already available in the followers ECL. From the figures presented in Table 1 we can deduce that the overheads for validation ($\lambda_{\text{OD}}$) of the two first implementations take up respectively 4.85 ms, and 5.27 ms, whilst the overheads for the asymmetric implementation take up only 2.02 ms.

We now analyse the overheads for the TMR nodes. Table 2 gives the average delays obtained when we executed the experiments on the different TMR node implementations.

<table>
<thead>
<tr>
<th>Model</th>
<th>Delay (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
</tr>
</tbody>
</table>
Table 2: Performance overhead for a client-server application on TMR nodes

<table>
<thead>
<tr>
<th>Model</th>
<th>RL (milliseconds)</th>
<th>RRPO (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-replicated node</td>
<td>5.74</td>
<td>0.00</td>
</tr>
<tr>
<td>fail-silent synchronised-time based</td>
<td>35.67</td>
<td>83.91</td>
</tr>
<tr>
<td>fail-silent time-out based</td>
<td>22.38</td>
<td>74.35</td>
</tr>
<tr>
<td>fail-silent asymmetric</td>
<td>14.46</td>
<td>60.30</td>
</tr>
<tr>
<td>TMR synchronised-time based</td>
<td>221.73</td>
<td>97.41</td>
</tr>
<tr>
<td>TMR time-out based</td>
<td>182.49</td>
<td>96.85</td>
</tr>
</tbody>
</table>
Table 3: Response latency and RRPO for a client-server application

In our experiment there is no processing at the server apart from that associated with queue manipulation. Thus, we can consider the figures presented in Table 3 as worst case performances for a lightly loaded node. It should be appreciated that the price in performance becomes significant in only these distributed applications where processes interact frequently. If on the other hand, application processes are involved in computations requiring less frequent interactions then the performance impact of replication protocols can be quite small. The next experiment illustrates this point.

Relative replication performance overhead for increasing server processing time

In this experiment we have extended the experiment previously described one step further. We have modified the server process in such a way that we can control its processing time. We then executed the experiment for different server processing times. Figure 6 shows the figures obtained for RRPO when the server processing time was increased from (almost) zero up to 100 ms.

Figure 6: RRPO versus server processing time

As expected, when the server processing time increases there is a reduction on the relative replication performance overhead of replicated nodes. The values shown in Figure 6 indicate that for our particular environment, if application processes do not communicate during intervals of time of at least 100 ms duration, then the burden associated with replication in the fail-silent nodes can be decreased from more than 80% to less than 25%. Also, for server processing times as little as 25 ms, the asymmetric fail-silent node can reach about 75% of the performance of a non-replicated node.

For the synchronised-time and time-out based TMR nodes, and server processing times of up to 100 ms, the decrease of the relative replication performance overhead is not as distinctive as was the case for the fail-silent nodes. However, the relative replication performance overhead of the early-order TMR node is reduced from more than 80% to nearly 30%. Given their more robust semantics, TMR nodes should naturally be expected to be more expensive to build.

Node delay versus message size

The next experiment was performed to evaluate the impact of the size of messages on the performance of a replicated node. On our system, the end-to-end message transmission delay between two transputers varied from 1.45 ms (messages of size 256 bytes) to 2.16 ms (messages of size 1536 bytes). Thus, the size of messages will affect intra-node message transmission times, consequently affecting both input and output delays.

Using the original client-server application previously described (i.e. with minimal processing at the server), we have measured the node delay for the various fail-silent and TMR nodes implemented, as the message size was increased from 256 to 1536 bytes. Figure 7 presents the average node delays obtained when the experiment was executed on the different replicated nodes implemented.

Figure 7: Node delay versus message size

The impact of the increase of message size on the performance of the various replicated nodes implemented is not uniform. For instance, the performance of the fail-silent nodes suffers only a relatively small decrease when compared with the reduction of performance experienced by the TMR nodes. The fail-silent node implementations require only a small number of intra-node messages to be exchanged between the processors forming the node. For these nodes, most of the increase on the node delay will be due to the extra time needed to authenticate, sign and copy messages within the node, rather than the increased transmission delay. This is confirmed by the values shown in Figure 7 which indicate an average increase of the node delay for the fail-silent node implementations of 19.41 µs/byte for the synchronised-time based implementation, 19.77 µs/byte for the time-out based implementation and 23.09 µs/byte for the asymmetric implementation.

For the TMR implementations, increasing the size of the messages impacts the performance of the nodes in a much more diverse fashion. The much larger internal message traffic of TMR nodes, particularly in the input phase, is responsible for a sharp increase of the node delay of the early-order
implementation (52.69 µs/byte on average). On the other hand, the synchronised-time based implementation suffers only a relatively small increase of 21.21 µs/byte on average. This is because the increased transmission delay will have no impact on the performance of the order protocol based on synchronised-time, since its stability delay is based on the worst case transmission delay. Finally, the node delay of the time-out based implementation suffers an intermediary increase of 28.40 µs/byte on average. Like the synchronised-time based implementation, part of the stability delay of this implementation is based on the worst case transmission delay, which reduces the rate with which the size of messages increases the node delay of this implementation.

Response latency versus number of clients

In the experiments so far, the server process executing on the replicated nodes attends a single request at a time. Thus, the figures obtained correspond to the case where the server process is subjected to a light load. We now analyse the behaviour of our system when the processing load is increased. We increase the load by increasing the number of clients that simultaneously issue requests to the server. The experiment is structured in such a way that a client process only issues another request to the server when all client processes have received the response of their previous request. Also, there is no processing at the server. Figure 8 shows the average response latency attained when the number of clients was increased from 2 up to 20.

Figure 8: Node delay versus number of clients

For the TMR implementations, the early-order node is the one that presents the largest increase in the response latency (24.03 ms/client on average). Both the synchronised-time and the time-out based TMR implementations present a much lower increase on their response latency as the number of clients increases (17.38 and 20.82 ms/client on average, respectively). The response latency for these two implementations is less affected because the processing time that would be normally wasted whilst the processor is idle waiting for an input message to become stable can now be used to attend requests from other clients. This behaviour can also be observed for the fail-silent implementations, although in this case the impact is not as noticeable as it was the case for the TMR implementations. The increase in the response latency of the synchronised-time based implementation is on average 7.35 ms/client, whilst the time-out based implementation has its response latency increased by 7.70 ms/client on average. The asymmetric fail-silent implementation still out-performs the other two implementations by a considerable margin, and has an increase on its response latency of 4.57 ms/client on average.

Under heavy load the nodes have their performance close to their respective worst case. Therefore, when the number of clients increases, for both the fail-silent and the TMR cases, the response latency of the time-out based nodes gets closer to the response latency of the synchronised-time based nodes. Indeed, for the TMR node implementations, when the number of clients is greater than 4, the response latency of the time-out based implementation is greater than that of the synchronised-time based implementation, reflecting the higher cost in maintaining timing counters.
5. Conclusions

We have investigated alternative ways of constructing efficient fail-controlled replicated nodes suitable for the development of a reliable processing platform, on top of which dependable distributed systems can be more easily implemented. The fail-controlled nodes presented in this paper are based solely on the utilisation of `off-the-shelf' processors (which can fail in an arbitrary way), and software protocols to control system redundancy, without recourse to any specialised hardware. We have implemented both three-processor failure-masking (TMR) and two-processor fail-silent nodes on a network of T800 Inmos transputers. Extensive experiments were performed to evaluate the performance of the nodes under various protocols for the ordering of input messages.

Most of the performance overheads of the TMR node implementations are associated with the ordering of input messages. In a lightly loaded, error-free system, the early-order implementation presents a much reduced performance overhead when compared with the performance of both the synchronised-time and the time-out based TMR implementations. For our particular system, the early-order implementation out-performs the synchronised-time based implementation by a factor of 4.6, and the time-out based implementation by a factor of 3.8. Reconfiguration mechanisms can be introduced to replicated nodes, so that they are expected to be in an error-free state for most of their lifetime.

Concerning the implementation of fail-silent nodes, the results obtained indicate that adopting the asymmetric mechanism for ordering of input messages within a fail-silent node represents the best design choice. Our performance figures have been obtained after quite a careful engineering of the message passing software. It is unlikely therefore that significantly better performance can be produced from soft fail-silent nodes. So the asymmetric node described here probably indicates the limits of what can be achieved using standard `off-the-shelf' processors. In our particular implementation, the performance impact of using fail-silent nodes is to impose an extra delay in response time of about 8 ms per message in a lightly loaded system. Further, when processes possess a communication intensive characteristic, a fail-silent node can achieve about 40% of the performance of its non-replicated counterpart. On the other hand, in those cases where application processes are involved in computations requiring less frequent interaction, then the performance impact of adding software-implemented fail-silence can be quite small. For our particular environment, the burden in the performance of an asymmetric fail-silent node can be reduced to less than 10% when the interval between two intra-process communication is larger than 100 ms.

Thus, bearing in mind the discussion on the advantages of soft replicated nodes over hard replicated nodes, we can anticipate a range of applications for which our soft replicated nodes offer an attractive alternative to their hardware implemented counterparts.

Acknowledgements

This work has been carried out in collaboration with the research team of the Voltan project at Newcastle University, and has been supported in part by grants from CNPq/Brazil. Comments from Neil Speirs, Paul Ezhilchelvan, Santosh Shrivastava and Sha Tao were helpful.

References


