Efficient Protocols for Fail-Silent Nodes in Distributed Systems

F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs and S. Tao

TECHNICAL REPORT SERIES

No. 413 February, 1993
EFFICIENT PROTOCOLS FOR FAIL-SILENT NODES IN DISTRIBUTED SYSTEMS

F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs and S. Tao

Abstract

A fail-silent node is composed of a number of "fail-arbitrary" conventional processors on which application level processes are replicated to achieve fault-tolerance. The non-faulty processors of a node need to execute agreement and order protocols to 'keep in step'. The design and the implementation of efficient protocols for a two processor fail-silent node are described in detail. In particular we introduce two new ordering protocols which substantially enhance the performance of the node.

© 1993 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Science, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

EFFICIENT PROTOCOLS FOR FAIL-SILENT NODES IN DISTRIBUTED SYSTEMS
Efficient Protocols for Fail-Silent Nodes in Distributed Systems
[By] F.V. Brasileiro
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.413)

Added entries
BRASILEIRO, Francisco Vilar
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Science, Technical Report Series. 413

Abstract
A fail-silent node is composed of a number of "fail-arbitrary" conventional processors on which application level processes are replicated to achieve fault-tolerance. The non-faulty processors of a node need to execute agreement and order protocols to 'keep in step'. The design and the implementation of efficient protocols for a two processor fail-silent node are described in detail. In particular we introduce two new ordering protocols which substantially enhance the performance of the node.

About the author
F.V. Brasileiro is a PhD student in Computing Science at the University of Newcastle upon Tyne.

P.D. Ezhilchelvan is Lecturer in Computing Science at the University of Newcastle upon Tyne.

S.K. Shrivastava is a Professor in Computing Science at the University of Newcastle upon Tyne.

N.A. Speirs is a Lecturer in Computing Science at the University of Newcastle upon Tyne.

S. Tao is a Research Associate in Computing Science at the University of Newcastle upon Tyne.

Suggested keywords
BYZANTINE AGREEMENT DISTRIBUTED PROCESSING FAIL-SILENCE
FAULT-TOLERANCE REPLICATED PROCESSING RELIABILITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 621.381958
U.D.C. 519.687 681.322.02
EFFICIENT PROTOCOLS FOR FAIL-SILENT NODES IN
DISTRIBUTED SYSTEMS

F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs and S. Tao
Computing Science Department, University of Newcastle upon Tyne, UK.

ABSTRACT
A fail-silent node is composed of a number of "fail-arbitrary" conventional processors on which application level processes are replicated to achieve fault-tolerance. The non-faulty processors of a node need to execute agreement and order protocols to 'keep in step'. The design and the implementation of efficient protocols for a two processor fail-silent node are described in detail. In particular we introduce two new ordering protocols which substantially enhance the performance of the node.

KEYWORDS
Byzantine Agreement, distributed processing, fault-tolerance, fail-silence, reliability, replicated processing.
1. INTRODUCTION

Replicated processing on distinct processors whereby outputs from faulty processors can be prevented from appearing at the application level (by employing means such as comparison or voting the outputs produced by the processors), provides a practical means of constructing systems capable of tolerating *fail-uncontrolled* processor failures. A node is composed of a number of conventional processors on which application level processes are replicated to achieve fault-tolerance in a manner suggested above. The basic idea is conceptually simple: a node is built out of several processors that execute special agreement and order protocols to carry out replicated processing of computations to achieve fault-tolerance. Such fault-tolerant nodes constitute the building blocks for constructing highly reliable distributed computing platforms. In this paper we will confine our attention to a node which either works correctly, or stops functioning (becomes silent) as soon as an internal failure is detected. Such a node has been termed a *fail-silent* node. We describe the implementation of a fail-silent node composed of two internal processors which implements the abstraction of fail-silent behaviour in the following sense: it either produces correct messages which can be verified as such by the receivers, or it ceases to produce new correct messages. This behaviour is guaranteed so long as no more than a single processor in the node fails. Any spurious messages produced by the failed processor of a node can be detected as such by all correctly functioning receiver nodes.

We assume a purely software approach to the management of redundancy and, further, that the nodes are to be used in a distributed system where processors as well as nodes communicate only by message passing. The implementation of fail-silent nodes presented here suggests that it is possible to construct nodes capable of tolerating a wide class of failures (Byzantine in the limit) by utilising standard 'off the shelf' components (processors), without recourse to any specialised hardware. In this respect, we have followed the approach, pioneered by the designers of the SIFT system [1], of using software implemented agreement and order protocols for supporting replicated processing. Unlike SIFT, our nodes are capable of supporting quite general purpose message passing application level programs.

We have designed and implemented failure-masking and fail-silent nodes. In [2] the detailed architecture of a family of such nodes is described, where we also discuss how distributed systems can be composed out of such nodes. More recently, we have performed a careful analysis of the performance of fail-silent nodes and have examined means of improving the performance. In particular we examine the ordering protocols
required by a fail-silent node and propose two novel methods of achieving order*. These techniques are shown to be considerably more efficient than the original protocols.

2. NODE ARCHITECTURE

2.1. System model and assumptions
We will assume that a failed processor (and therefore the processes running on that processor) can exhibit fail-uncontrolled behaviour. We assume that (non-replicated) distributed computations have been composed out of a number of processes that interact only via messages. As an example, the function of a typical 'server' process is to repeatedly pick up an input message from 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 on those 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 will eventually select a message present on a port. The model presented here is based on the well known state machine model (where a state machine is a process) for which the precise requirements for supporting replicated processing are known [3]. It is also necessary to assume that the computation performed by a process on a selected message is deterministic. Given such a model of computation, active replication of a process will require the following two conditions to be met:

Agreement: All the non-faulty replicas of a process receive identical input messages;  
Order: all the non-faulty replicas process the messages in an identical order.  
So, if all the non-faulty replicas of a process of a node have identical initial states then identical output messages will be produced by them. This is the underlying principle of active replication. Practical distributed programs often require some additional functionality 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 previous papers [2, 4] we have described how our nodes provide the necessary functionality for dealing with such cases. In this paper, we will assume the simple state machine model discussed above.

2.2. Node Architecture
The overall node architecture is such that each of the two processors has a network interface for reliable inter-node communication over (possibly redundant) networks; in addition, the processors are internally connected by a communication link for intra-node communication needed for clock synchronisation and order protocols. Each non-faulty

* Patent applied for.
processor in a node is assumed to be able to sign a message it sends by affixing the message with its (the processor’s) unique, message dependent, unforgeable signature; it is also assumed to be able to authenticate any signed message it receives, thereby detecting any attempts to corrupt the message. Digital signature based techniques [5] provide such functionality with high probability. To achieve agreement and order, the non-deterministic selection of a message by a process is now replaced by selection of the message at the head of a single message queue (called the delivered message queue) of a replica, and ensuring that the queues of the replicas will contain identical messages in an identical order.

Each non-faulty processor of a node has five processes:

a) **Sender Process**: this process takes the messages produced by the computational processes of that processor, signs them and sends them via the link to the neighbour processor of the node for comparison.

b) **Comparator Process**: this process authenticates all incoming messages from the neighbouring processor; an authenticated message \( m_i \) is compared with its counterpart produced locally. If the comparison succeeds, the authenticated message \( m_i \) is counter signed (by considering the first signature as a part of the message) and this double signed message is handed over to a *transmitter process* for network delivery to destination nodes. A locally produced message that cannot be compared because its counterpart does not arrive within a specified time-out period or a comparison that detects a disagreement indicates a failure. Once a failure is indicated, the comparator process stops, which results in no further double signed message being produced from that node.

c) **Transmitter Process**: this process is responsible for sending the double signed messages over the network to destination nodes.

d) **Receiver Process**: this process authenticates messages received from the network or the link for processing and discards any duplicate messages it receives. Authenticated messages from the network are sent to the local *order process*. Singly signed messages from the link are stored for comparison with locally produced messages.

e) **Order Process**: this process executes the order protocol described below with its counterpart in the other processor and attempts to construct identical queues of valid messages for processing by the computational processes.
An implementation of the order protocol will require that the clocks of both the processors of a node are synchronised such that the measurable difference between readings of clocks at any instant is bounded by a known constant $\varepsilon$. Algorithms for achieving this abstraction exist (see [6, 7]). Communication for clock synchronisation and ordering takes place by way of an internal link between processors of the node. Given that the clocks of both the processors are synchronised, the order protocol is particularly simple, since it is expected to work only in the absence of any failures. Essentially, an order process of a processor stamps a message to be ordered with its local clock reading. A copy of the time-stamped message is signed and sent over the link to the order process of the other processor in the node. If $T$ is the timestamp of the message received from or sent to the order process of the other processor, then the message becomes stable at local clock time $T+\Delta = T+\delta+\varepsilon$ where $\delta$ is the maximum transmission time taken for a time-stamped message to travel from one order process to another order process over the link. Once a message with timestamp $T$ becomes stable, no message with timestamp $T' < T$ can be received by an order process. Stable messages are queued in the delivered message queue in increasing timestamp order (with care being taken not to queue a stable message, if its replica has already been queued).

A correctly functioning node will generate two identical copies of its output messages. A receiver process at a node will discard any duplicate messages received over the network. When the comparator process of a non-faulty processor in a node detects a failure and therefore stops, no new, double signed messages can be emitted by the node; any messages coming from this node that are not double signed will not be found to be authentic at the receiving nodes. Any authentic but old messages from a faulty node will also be discarded by the receiving nodes as replicas of already received messages.

The overall software architecture of a node is depicted in Figure 1, where the major components of the system within a processor of a node and their interactions are summarised. For the purpose of storing valid messages, a processor maintains several message queues:

a) Received Message Queue (RMQ): Contains valid received messages intended for ordering.

b) Processed Message Queue (PMQ) Contains unsigned output messages produced by local computational processes. These messages must be validated: checked by the comparator before transmission to the final destination.
c) External Candidate Message Queue (EMQ): Contains singly signed messages that have been received for validation.

d) Internal Candidate Message Queue (IMQ): Contains unsigned messages, each waiting for a matching signed message to arrive in EMQ.

e) Delivered Message Queue (DMQi). Contains ordered messages to be consumed by the process Servicei.

Figure 1: Software architecture of a processor in a node

The order process is composed of three cyclic processes: relayer, transfer and deliver. The Relayer process picks up messages from the RMQ, timestamps them and sends...
them to the other processor in the node. It also inserts the message into the ordered message list (OML) in increasing timestamp order (if two distinct messages contain identical timestamps, they are ordered by their processor numbers). The Transfer process picks up relayed messages from the Transferred Message Queue (TMQ), performs a timeliness check and inserts accepted messages into the OML. The Deliver process takes stable messages from the head of the OML, removes duplicates and enqueues the messages on the appropriate DMQi.

2.3. Node Fault Coverage Analysis

In [8], a convenient classification of the possible faults that can affect a system is presented. In that paper, correct behaviour of the components of a system is analysed both in the value and time domain. A correct output not only has the correct value (according to specification) but is also produced at an acceptable time that is neither early nor late. Faults are classified as follows:

A fault that causes a component not to respond to a request is termed an **omission fault** (note that an omission fault can be transient, e.g. a communication link can occasionally lose a message and continue to work properly afterwards);

A fault that causes a component to output erroneous values is termed a **value fault**;

A fault that causes a component to respond to a request either early or late is termed a **timing fault**;

A fault that causes a component to output incorrect responses in both the value and timing domains is termed an **emission fault**; and finally

A fault that causes a component to violate its specified behaviour in any manner (e.g. producing an unexpected output) is termed a **Byzantine fault**.

The fail-silent node presented here exhibits fail-silent behaviour so long as no more than one of the processors (which may well contain Byzantine faults) fails.

We begin our fault coverage analysis by supposing that communication is perfect and failures are restricted only to processors (an inter processor communication failure will be attributed to one of the processors involved). Correct messages are signed by both processors in a node so that, given our signature assumptions, we can state that a faulty processor will not be able to output a message containing two distinct and authentic signatures on its own. The comparison mechanism of a non-faulty processor can detect any erroneous message produced by the faulty neighbour. Thus, all successfully compared and double signed messages output are correct in the value domain. As soon
as a disagreement is detected, the non-faulty processor stops its activities and no more double signed messages will be output by that node. Receivers at a destination node can detect corrupted messages whilst authenticating all incoming messages received from the network.

Failures in the time domain can only be detected if there is an event (e.g. the reception of a message) that must occur within a pre-defined interval of time. If a processor suffers a timing failure, no difficulty arises if the fault is manifest after it has sent the expected message to the other processor of the node. The latter will receive a valid message and will be able to output a correct double signed message. If a processor does not receive a counterpart message from its neighbour (i.e. the fault manifested before the exchange of intra-node messages), the non-faulty processor can assume that its neighbour is faulty and stop its own activities. As a consequence, no further double signed messages will be output from the node. The only difficulty occurs with messages that have already been sent to a faulty (in the time domain) neighbour within the time between the fault manifestation in the faulty processor and the fault detection by its companion. The faulty processor is able to issue untimely (late) double signed messages.

Another problem is that of a 'babbling' processor i.e. a faulty processor which sends out many invalid messages and floods the network. Both of the above problems can be contained if processors within node can unilaterally switch off the interface to the network. Any processor which detects an error in the node first switches off the network interface and then halts its processing. If a faulty processor closed down the interface to the network, this would not matter since, by definition, the node is faulty. A simple switch of this nature would be straightforward to implement.

3. NODE PERFORMANCE

In a fail-silent node the order protocol is responsible for much of the performance overhead of the replicated system. Every message received by one processor is relayed to its neighbour. A non-faulty processor knows that after $\Delta = \delta + \varepsilon$ time units have elapsed (as measured by its clock), the message sent to the neighbour will have been received and placed in the correct place in the ordered message list. The fixed overhead implicit in this method has motivated us to seek enhancements to the ordering protocol. We have designed two new protocols for ordering messages which are discussed below.
3.1. Reducing the Message Stability Delay

The arrival of a relayed message in one processor can be used to reduce the constant stability delay $\Delta$ imposed by the order protocol on previously received messages. We shall assume that each processor generates monotonically increasing timestamps and also that incoming messages (in RMQ) are relayed to a processor’s neighbour according to a First In First Out (FIFO) policy. Given this assumption, the timestamp of a received relayed message defines intervals of time where messages can be stabilised earlier than the time given in section 2.1. Figure 2 illustrates this property.

![Timestamp of Relay Message Diagram](image)

In case (a) a relayed message with timestamp smaller than the local time is received. As no more messages will be generated with timestamps smaller than the timestamp of the relayed message (timestamps assigned to new local messages will be greater than the local time and new relayed messages will necessarily have timestamps greater than that of the relayed message just received) all previously received messages, local or remote, with timestamps smaller than or equal to the timestamp of the relayed message are stable.
In case (b) a message with timestamp greater than the local time is received. In this case it is certain that no more messages will be generated with timestamp smaller than the local time, thus every previously received message with timestamp smaller than or equal to the local time is stable.

Hence the set of stable messages \( S \) is defined by:

\[
S = \{ m_T : T \leq \max(\text{Local Time} - \Delta, \min(\text{Local Time, Last Relayed Timestamp})) \}
\]

We can state that in this protocol, the time taken to stabilise a message is bounded by \( \Delta \), so that in the worst case we shall obtain the performance of the order protocol described in section 2.1. If required, processors in the fail-silent node may send dummy message regularly to each other so that the stability delay may be bounded by a value smaller than \( \Delta \).

3.2. Asymmetric Ordering Protocol

In this protocol, we assign different roles for each of the processors forming a node. We will term one processor the leader and its neighbour the follower [9]. It is the responsibility of the leader to determine the order of processing messages. Having selected a message for processing, the leader sends a copy of the message to the follower. The internal structure of the node, depicting message flows, is shown in Figure 3. Here we have used the same names for the various message buffers as in the original architecture of Figure 1.

The node works as follows; the leader will receive messages from the network and sends a copy of them in the order in they should be consumed to the follower. After transmission of the request message to the follower, the leader processes the request and afterwards sends a signed result message to the follower where this message will be deposited in the External Message Queue (EMQ). Meanwhile, the follower processes messages in the order they are received from the leader (the order in which the follower receives messages from the network is not relevant). When the follower has processed a request, it deposits the result message in its Internal Message Queue (IMQ). The follower then compares the leader's response from EMQ with its own response in IMQ. If the two responses agree, the message from EMQ is counter-signed and then sent to the destination, otherwise, the follower process stops its activities and no further double signed messages will be output. After a successful comparison, the follower also transmits the result message to the leader's EMQ. The leader then compares and transmits the double signed message.
Figure 3. Leader/Follower Fail-Silent Node.

The Service, Compare and Send processes work in almost the same way as in the normal Fail-Silent architecture. The Rx_Int and Tx_Int receive and transmit messages within the node. The Rx_Ext and Tx_Ext receive and transmit messages to other nodes in the system. The Neighbours Message Queue (NMQ) is used to store messages which are to be sent to a processor's neighbour and the Compared Message Queue (CMQ) holds compared messages awaiting transmission.

The asymmetry introduced by assigning different roles to the two processors of a node requires us to introduce extra mechanisms in order to detect processor failures. A Timing process in the follower is introduced in order to detect omission and timing failures occurring in the leader and also the non-arrival of request messages (double signed messages from other nodes) at the leader. In a correctly functioning system, both processors will receive the same request messages from outside the node (although not necessarily in the same order). The follower's Rx_Ext process receives each request from outside the node and deposits it in the External Received Message
Queue (ERMQ) with an associated time-out. The Timing process picks up each message in the Internal Received Message Queue (IRMQ) and resets the time-out associated with its counterpart in the ERMQ. If a time-out expires, the follower may assume that the leader has failed and so the follower will cease its own activities. As a result, no more double signed messages will be produced. Alternatively the follower may try to mask the failure (it may be the case that the leader never received a request message) by sending its copy of the request message to the leader.

Detecting omission and timing failures in the follower is achieved by the follower sending to the leader, the single signed result messages. After comparing the follower's message against its own output, the leader will also output a double signed message if there are no discrepancies. If the message from follower to leader does not arrive in a 'reasonable' time, the leader will stop sending messages to the follower, and so no more double signed messages will be produced.

To calculate the time to process a request for this node, it is necessary to consider the activities in both processors. As the activities in the leader and follower processors execute in parallel, we cannot simply add the times of the activities executed in each node. The Service processes are executed in both processors in parallel. However, the follower has to wait for the request message sent by the leader before service can begin. The Compare process in the follower processor has to wait for the messages produced by both leader and follower. The leader begins to service requests before the follower so the Compare process in the follower will seldom be delayed in waiting for the leader's response message. Since processors in a fail-silent node must exchange at least one message per request (the message to be compared) we should expect the leader/follower fail-silent node to have near optimal performance for a software implemented fail-silent node.

4. IMPLEMENTATION

Instances of the node architecture described above have been implemented. They have been developed on a network of T800 Inmos Transputers to produce two processor Fail-Silent nodes with the 3 different ordering protocols discussed earlier (for a discussion of the implementation of a 3-processor Failure Masking node see [10]). The two processors of a node are directly connected to each other by transputer links, thereby providing an internal path for intra-node communication. Elsewhere we have described how such nodes can be used as building blocks for constructing reliable distributed computing platforms [1, 10-12]. The software implementing the replication
and communication layers has been written in C++ [13] and implemented on the Helios operating system [14], a Unix like operating system which runs on transputers and supports the client/server model for structuring application programs.

4.1. System Services
In Helios, processes are spawned by a call to the Fork() function. In our system we encapsulate process spawning into a common Active_Object base class from which individual process classes may then be derived. In this way, the base class provides the thread of activity while the derived class provides the algorithm by specifying the behaviour of the virtual operation main(). A parameter supplied to the derived class constructor determines the priority of the thread. There are two priorities available, high and low, corresponding to the priorities handled by the transputer scheduler.

Two active objects can communicate with each other by accessing a common passive object via a pointer parameter passed to both constructors. Note that any implementation of a passive object class must consider issues of concurrency control. The Queue class allows concurrent active objects to deposit() and remove() without interference. All the message pools and queues required are declared as instances of a class (Message_Block_Queue) which is derived from the base class Queue.

A message is implemented as a passive object Message_Block and is defined as a class which represents the structure of a message accepted by queues and transmitted across transputer links. A Message_Block contains a control component which handles all the system information relating to a message (for example, signatures, sequences numbers, timestamps etc.). Message blocks also store message data in the form of a sequence of bytes. Hence the structure of the system consists of several processes (active objects) communicating asynchronously with each other via queues (passive objects) using message blocks (passive objects) to transfer information.

4.2. Fail-Silent Node Performance
In this section we give performance figures for a lightly loaded two processor fail-silent node. The clock synchronisation algorithm of Halpern et al. [6] gives the maximum difference between the clocks of correct processors $e$ as:

$$e = t_{\text{del}} + R\rho$$

where $t_{\text{del}}$ is the maximum delay in sending a clock synchronisation message between processors in a node, $R$ is the interval between resynchronisation and $\rho$ is the rate at which clocks drift. Terms of $O(\rho^2)$ and $O(\rho t_{\text{del}})$ have been neglected in the above equation. In our system we estimate that $t_{\text{del}} = 3$ms and take $R = 5$s. Taking $\rho = 10^{-6}$,
the second term in the above equation becomes negligible and we obtain a clock synchronisation $\epsilon$ of 3ms.

Currently, simple checksums are being used as signatures and so have a minimal impact upon system behaviour. The impact of using more complex signature mechanisms has not yet been assessed.

We now analyse the performance of the system with the three different ordering strategies described earlier. We consider a system where our fail-silent node receives request messages from a client, services these requests and sends back results. We measure the following time intervals:

(i) Input delay (ID) = remove($m_1$, DMQi) - $m_1$ received
(ii) Waiting delay (WD) = remove($m_1$, DMQi) - deposit($m_1'$, OML)
(iii) Service delay (SD) = deposit($m_2$, PMQ) - remove($m_1$, DMQi)
(iv) Output delay (OD) = $m_2$ sent - deposit($m_2$, PMQ)
(v) Total delay (TD) = $m_2$ sent - $m_1$ received = ID + SD + OD
(vi) Node delay (ND) = ID + OD = TD - SD

The Input delay measures the time between a message ($m_1$) entering the processor and it being removed from the Delivered Message Queue for processing. It reflects the overhead involved in ordering messages. The Waiting delay is part of the Input delay. It indicates the time which the message ($m_1'$) relayed from the neighbour spends in the Ordered Message List awaiting stability. The Waiting delay is only measurable for the non leader/follower implementations. The Service delay measures the time taken to service a request and put the result message ($m_2$) into the Processed Message Queue. The Output delay measures the time between a message entering the Processed Message Queue and the message being sent from the node. It reflects the time taken for messages to be compared. The Total delay is simply the sum of the input, service and output delays. The Node delay removes the service time from the Total delay i.e. it reflects the overhead associated with replication. In order to try to minimise the effects of scheduling delays, all processes were run at high priority. Clearly it is not possible to do this in a realistic system. However, it allows us to analyse the true overheads of the protocols employed. When run at low priority, the Total delays were not significantly affected though some of the intermediate timings were sometimes artificially changed by the context switching as a result of the transputer's scheduler.

The various delays are presented in Table 1 for trials of 1000 message requests.
<table>
<thead>
<tr>
<th>Model/Delay (msecs)</th>
<th>Input</th>
<th>Waiting</th>
<th>Service</th>
<th>Output</th>
<th>Total</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unreplicated System</td>
<td>0.36</td>
<td>0</td>
<td>0.77</td>
<td>0.37</td>
<td>1.50</td>
<td>0.73</td>
</tr>
<tr>
<td>Order Protocol</td>
<td>10.25</td>
<td>6.34</td>
<td>0.98</td>
<td>4.60</td>
<td>15.83</td>
<td>14.85</td>
</tr>
<tr>
<td>Optimised Ordering</td>
<td>5.20</td>
<td>0.58</td>
<td>1.05</td>
<td>4.93</td>
<td>11.18</td>
<td>10.13</td>
</tr>
<tr>
<td>L/F (Leader)</td>
<td>0.83</td>
<td>0</td>
<td>0.77</td>
<td>5.25</td>
<td>6.85</td>
<td>6.08</td>
</tr>
<tr>
<td>L/F (Follower)</td>
<td>1.67</td>
<td>0</td>
<td>0.78</td>
<td>2.00</td>
<td>4.45</td>
<td>3.67</td>
</tr>
</tbody>
</table>

Table 1. Performance of Fail-Silent Nodes

First we consider the behaviour of an unreplicated service. As we would anticipate, the average Node Delay in this case is small. It exists because the unreplicated node still enqueues and dequeues messages in the system.

In the standard order protocol the stability delay $\Delta = \epsilon + \delta$ was set at 9ms. The clock synchronisation $\epsilon$ is 3ms as discussed above. Experiments determined that the smallest safe value for $\delta$ to be 6ms. The stability delay is largely responsible for the sharp rise in the node delay when compared with the unreplicated results. Using the optimised ordering protocol with the stability test given in section 3.1, we should expect the stability time to reduce to the average time to transmit a message (approximately 3ms) i.e. a total reduction of 6ms. The results given in Table 1 show that this theoretical expectation is almost realised in practice. The Waiting time is reduced by 5.8ms and the Node delay by 4.7ms. Note that the Service times for the Order and Optimised Ordering protocols are larger than in the unreplicated case. This is caused by the deliver process in the order protocol pre-empting the service in order to check whether any messages can be delivered to a process.

For the Leader-Follower ordering mechanism discussed in section 3.2., it is necessary to examine the performance of both leader and follower since they are executing different protocols. The input delay in the follower is defined to be the time between receiving $m_1$ at the Leader and remove($m_1$, DMQ$_i$) at the follower. Hence it reflects the time taken for the leader to receive the message, relay it to the follower and have the follower remove it from DMQ$_i$. The service times for both leader and follower are the same as those for the protocols which order inputs (the internal scheduling of the transputers is responsible for the slight variance in service delays). However, the output delays are significantly different. In the follower, the output delays are smaller than in the leader because the leader begins to service the request before the follower so that when the follower is ready to compare its result, the leader will have already sent (or be sending) its response. If the comparison at the follower is successful, the follower outputs the compared message before passing its response to the leader.
Hence the output delay at the leader reflects this additional time. If the system is lightly loaded, its response time will be the Total delay as measured by the follower. In a heavily loaded system where the input queue to the leader always contains a message, the Total delay in the system will be that given by the leader. In either case, the performance of the leader/follower fail-silent node is considerably better than either of the fail-silent nodes employing an order protocol. If the application services involve lengthy computations then the percentage overhead involved in adding replication can be quite small. It is only when communication time between nodes outweighs computation time within nodes that the cost of replication becomes significant.

5. CONCLUDING REMARKS
We have described our initial experiments in building fail-silent nodes using a variety of ordering protocols. The experimental figures in section 4 indicates that adopting the Leader-Follower mechanism within a fail-silent node leads to a significant improvement in performance. The overhead of using fail-silent nodes is to produce a delay in response of approximately 3.7ms in a lightly loaded system (the Follower Node delay) up to 6.1ms in the worst case (the Leader Node delay). These figures have been obtained by using careful implementation techniques. It is unlikely that significantly better performance can be produced from fail-silent nodes and the leader/follower node described here probably represent the limits of what can be achieved using standard 'off-the-shelf' equipment.

ACKNOWLEDGEMENTS
This work has been supported in part by grants from the UK Science and Engineering Research Council and CNPq/Brazil.
REFERENCES


