IMPLEMENTING FAIL-SILENT NODES FOR DISTRIBUTED SYSTEMS

F.V. Brasileiro[1], P.D. Ezhilchelvan[2], S.K. Shrivastava[2], N.A. Speirs[2] and S. Tao[3]

[1]Departamento de Sistemas e Computacao,
Universidade Federal da Paraiba, Campus II,
58.109-970, Campina Grande, Paraiba, Brazil.

[2]Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK.

[3]Parallax Solutions Ltd.,
University of Warwick Science Park,
Coventry, CV4 7EZ, UK.

ABSTRACT

A fail-silent node is a self-checking node that either functions correctly or stops functioning after an internal failure is detected. Such a node can be constructed from a number of conventional processors. In a software-implemented fail-silent node, the non-faulty processors of the node need to execute message order and comparison protocols to 'keep in step' and check each other respectively. In this paper the design and implementation of efficient protocols for a two processor fail-silent node are described in detail. The performance figures obtained indicate that in a wide class of applications requiring a high degree of fault-tolerance, software-implemented fail-silent nodes constructed simply by utilising standard 'off-the-shelf' components are an attractive alternative to their hardware-implemented counterparts that do require special-purpose hardware components, such as fault-tolerant clocks, comparator and bus interface circuits.

Index Terms

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 comparing or voting the outputs produced by the processors), provides a practical means of constructing systems capable of tolerating Byzantine (also referred to as *fail-uncontrolled*) processor failures. Such an approach can be used for constructing a *fail-controlled* node composed of a number of conventional processors on which application level processes are replicated. A particular case of a fail-controlled node is a π+1 processor *fail-silent node* that either works correctly, or stops functioning (becomes silent) soon after an internal failure is detected. This behaviour of a node is guaranteed so long as no more than π processors in the node fail. A two processor fail-silent node (π=1) offers a practical and economical solution to the problem of constructing fail-controlled nodes, as such, in this paper we will concentrate on the design, implementation and performance evaluation of two-processor nodes. In particular, we will describe practical designs of software implemented two-processor fail-silent nodes suitable for use in distributed systems that meet the abstraction of fail-silence in the following sense: a node produces either correct messages which can be verified as such by destination nodes, or it ceases to produce new correct messages, in which case destination nodes can detect any messages it may produce as unwanted.

The paper is structured as follows. We begin by reviewing related work in the area of reliable node design, contrasting it with our approach and summarising the main contributions of the paper. We then describe the basic principles that underpin our fail-silent nodes, and then present what we term a *reference implementation* of a fail-silent node; this implementation makes use of a standard, synchronised clock based message order protocol. After describing how the performance of this protocol itself can be improved, we present two new, much faster order protocols, based on logical clock and leader-follower (master-slave) approaches. Following this, we describe the design of a comparison protocol that makes use of the master-slave approach for message comparison. We then present the results obtained from our experimental work on comparative performance evaluation of the various implementations of the fail-silent nodes; conclusions from our work are presented in the final section of the paper.

2. RELATED WORK

A fail-controlled node that uses replicated processing with comparison/voting must incorporate mechanisms to keep its replicas synchronised, so as to avoid the states of the replicas from diverging. Asynchronous events (e.g., interrupts, timeouts), processing of non-identical messages are some of the reasons that could lead to replica state divergence.

Synchronisation at the level of processor micro-instructions is logically the most straightforward way to achieve replica synchronism. In this approach, processors are driven by a common clock source which guarantees that they execute the same steps at each clock pulse (of course, the logic of the individual processors must be deterministic). Outputs are evaluated (compared/voted) by a -possibly replicated- hardware component at appropriate times (e.g., at each bus access). Asynchronous events must be distributed to the processors of a node through special circuits which ensure that all the correct processors will perceive such an event at the same point of their instruction stream [12, 23]. Since every correct processor of a node executes the same instruction stream, all the programs that run on the non-redundant version can be made to run, without any changes, on the node. This is the major advantage gained by synchronising at the level of micro-instructions. Such implementations of two processor fail-silent nodes have been in use widely; Stratus [27] and Sequoia [2] are two well-known examples. In these systems, a common (reliable) clock source is used for driving a pair of processors which execute in lock-step. Access to the bus is controlled by a (reliable) comparator circuit which only enables access to the bus if the signals generated by the two processors are the same. Another example of a fail-controlled node is presented in [6]; this design employs tight synchronisation of redundant processors and in addition, uses coding techniques for detecting/correcting memory bit corruptions.

There are however a few problems with the micro-instruction level approach to synchronisation. First, as indicated before, individual processors must be built in such a way that they will have a deterministic behaviour at each clock pulse, so that they will produce identical outputs ("don't care" transitions, for instance, where a bit can be either one or zero, are not allowed in the design of the processors). Second, the introduction of special circuits such as reliable comparator/voter, reliable clock, asynchronous event handlers, and bus interfaces, increases the complexity of the design, which in the extreme can lead to a reduction in the overall reliability of a node. Third, every new microprocessor architecture requires a considerable re-design effort. Fourth, because of their tight synchronism, a transient fault is likely to affect the processors in an identical manner, thus making a node susceptible to common mode failures.

Approaches that do not use processor replication but rely instead on various *application specific* forms of checking mechanisms (e.g., watchdog timers) for detecting the erroneous behaviour of a processor have therefore been considered [e.g., 17]. The error detection coverage of one such node has been estimated to be better than 99% [11]. However, these approaches are application specific (rather than general purpose) and do not completely eliminate the second and third problems referred to above.

An alternative approach that seeks to reduce (or eliminate altogether) the hardware level complexity associated with the approaches discussed above is to maintain replica synchronism at a higher level, for instance at the process, or task level by making use of appropriate software implemented-protocols. Such software-implemented nodes can offer several advantages over their hardware-implemented counterparts: (i) technology upgrades appear to be easy; since the principles behind the protocols do not change, the protocol software can be ported relatively easily to any type of processor (including the ones expected to be available in the future); (ii) we note that by employing different types of processors within a node, there is a possibility that a measure of tolerance against design faults in processors can be obtained, without recourse to any specialised hardware assistance; and (iii) since replicated computations do not execute in lock-step, a node is likely to be more robust against transient failures [11].

The task synchronisation approach was pioneered by the designers of the SIFT failure-masking node [28]. In SIFT, application processes are structured as a set of co-operative cyclic tasks. Each task performs a deterministic computation. The execution of a particular iteration of a task consists of inputting some data (possibly generated by previous iteration of other tasks), processing the data, and outputting some results. Fault-tolerance is achieved by voting on the input data. Thus, task replicas must be synchronised at the beginning of each iteration (start of a *frame*). To achieve this, SIFT maintains a global timebase, and uses a static, priority based scheduling, which schedules tasks at pre-defined time frames. The global timebase is implemented by keeping the clocks of all the correct processors synchronised by a software implementation of a Byzantine resilient clock synchronisation protocol. In normal operation, the system only allows interruptions from clocks, which are handled by all correct processors at the beginning of the same time frame. Because of its application dependent design, the SIFT architecture can only be applied to a restricted range of applications. This is also the case for the VOTRICS system [25] which follows the design principles of SIFT to provide fault-tolerance in a different, but still specific, class of applications (railway signalling systems).

In our work, we have taken the SIFT approach further by investigating the design of a family of failure-masking and fail-silent nodes (called Voltan [21, 22, 24]) that are capable of supporting quite general purpose message passing programs. Voltan nodes are composed of 'off-the-shelf' processors connected via communication links. The processors of a node execute message agreement and ordering protocols to guarantee that correct replicas of application processes will receive and process input messages in identical order. The output messages produced by process replicas are evaluated either by a comparator (a fail-silent node), or a voter (a failure-masking node) at each processor.

There is however a concern over the performance of software-implemented nodes due to the overheads imposed by redundancy management protocols. Indeed, in terms of performance, hardware-implemented nodes will always out perform their software equivalents (a hardware-implemented node will be capable of working at nearly the same speed as its constituent processors). In SIFT for instance, redundancy management protocols can consume as much as 80% of the processor throughput [15]. Hybrid solutions have been proposed to circumvent this problem. MAFT [10], FTP-AP [13], and Delta-4 [16] are hybrid architectures that share the same basic design. These architectures are structured around a micro-instruction synchronised hard core, on top of which conventional processors are replicated. The micro-instruction synchronised hard core is responsible for executing redundancy management functions (e.g., message voting). This certainly improves the performance; however, the hard core re-introduces the problems associated with the hardware-implemented nodes.

In this paper we present the design and implementation of software-implemented two-processor fail-silent nodes that are both efficient (in terms of performance) and capable of executing general purpose message passing software. We have performed a careful analysis of the performance of our original implementation of Voltan nodes (the reference implementation) and have examined several ways of improving its performance. This has led to the design of two novel message order protocols which are considerably more efficient than the original protocol. A property of a fail-silent node that has been exploited in our design for obtaining efficiency is that it is required to just *detect* a failure rather than *mask* it. We present these protocols and the resulting performance of the nodes. The performance figures obtained lead us to believe that in a wide class of applications requiring a high degree of fault tolerance, software implemented fail-silent nodes constructed simply by utilising standard 'off-the-shelf' components and employing one of the new order protocols (particularly the leader-follower protocol) do represent an attractive alternative to their hardware implemented counterparts.

3. BASIC PRINCIPLES

3.1. System Model and Assumptions

We assume that a failed processor (and therefore the processes running on that processor) can exhibit Byzantine behaviour; but we do make the assumption that each non-faulty processor in a node is able to *sign* a message it sends by affixing the message with a message dependent unforgeable signature; a non-faulty processor is also assumed to be able to *authenticate* any signed message it receives. Digital signature based techniques [18] provide a very comprehensive way of meeting this functionality. We assume that non-replicated distributed computations are composed of a number of processes that 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. It is necessary to assume that the computation performed by a process on a selected message is *deterministic*. This is the well known *state machine* model (where a state machine is a process) for which the precise requirements for supporting replicated processing are known [20]. Basically, in the replicated version of a process, 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. So, if all the non-faulty replicas have identical initial states then identical output messages will be produced by them, provided the queues of all correct replicas can be guaranteed to contain identical messages in an identical order. Thus, replication of a process requires 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.

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 [22, 26] we have described how our nodes can be enhanced to provide the necessary functionality for dealing with such cases. In this paper, we will assume the simple state machine model discussed above.

We assume that each processor of a fail-silent node has 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 (e.g., message ordering and comparison). We assume that the maximum intra-node communication delay over a link is known and bounded: if a non-faulty process sends a message over a non-faulty link to a non-faulty process of a neighbour processor then the message will be received within δ time units. For simplicity, we will assume that the lower bound on the actual transmission delay, δ_a, is zero: $0 \le \delta_a \le \delta$ (so δ also represents the maximum variation in message transmission delays over a link). Link failures will be categorised as processor failures: a link failure that prevents a message sent from a processor to be received by its neighbour in the node will be considered as a failure of the sender processor.

Fig. 1 shows an example of a distributed system with three two-processor fail-silent nodes (P, S and Q), connected by a dual redundant network (C1, C2). On such an architecture, 'node level' processes can be replicated on distinct nodes for increased availability (a node level process itself is composed of two processes, one on each of the underlying processors, and behaves like a fail-silent process). In particular, such a system architecture can be used for building *highly available services* by constructing *K-resilient* node processes: a K+1 replicated node level process (K>0) can tolerate a maximum of K replica failures before a subsequent failure makes the services it is providing becoming unavailable. In a separate paper we have shown how protocols for group communication between node level processes, necessary for supporting such services, can be implemented to run on two processor fail-silent nodes [7].

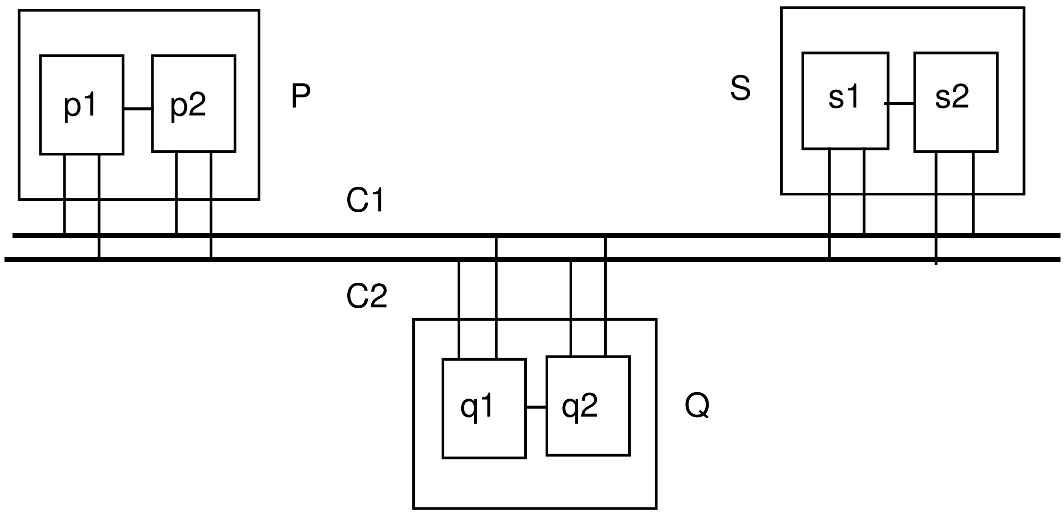

Fig. 1: A distributed systems architecture employing fail-silent nodes

3.2. Basic Software Architecture

We now describe the basic software architecture of a two-processor fail-silent node. In addition to application level computational processes, each processor of a node executes five *system processes* described below:

i) 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.

ii) Comparator Process: this process compares authentic messages sent by the neighbour processor with their counterparts produced locally. If a message comparison succeeds, the singly signed authentic message received from the neighbour is counter signed (by considering the first signature as a part of the message) and this double signed message, termed a *valid message*, is handed over to the local Transmitter process for network delivery to destination nodes. A comparison that detects a disagreement indicates a failure. Similarly, an absence of a message for comparison (after a node specific time-out interval) also indicates a failure. Once a failure is detected, the comparator process stops, and so does the sender process. No new valid messages can be produced by the node.

iii) Transmitter Process: this process is responsible for sending the double signed messages over the network to destination nodes. As each processor has a Transmitter process, a node with correct processors will produce two copies of its every output message. In our subsequent discussions on timing analysis of a node, a node output will refer the valid copy that is produced first.

iv) Receiver Process: this process authenticates messages received from the network or from the link and discards any unauthentic or duplicate messages. Authenticated messages from the network (valid messages) are sent to the local Order process**.** Authenticated singly signed messages from the link are sent to the Comparator.

v) Order Process: this process executes an order protocol with its counterpart in the other processor of the node in order to construct identical queues of valid messages for processing by the computational processes. Since such a protocol entails the Order process to relay valid messages to its counterpart, it is sufficient for a message to be received from the network by any one of the processors of a node for it to be ordered at both the processors (the only exception is the asymmetric order protocol without feedback, to be discussed later, which requires a message to be received by a nominated processor -the leader- for ordering).

The architecture can be adapted for the more general case of π+1 processor fail-silent node; such a node will produce valid messages with π+1 processor signatures.

3.3. Node Failure Semantics

We assume that application processes of correctly functioning nodes assign monotonically increasing sequence numbers to new messages they produce; this property enables correctly functioning destination nodes to discard replicas of any previously received messages. Let an application process running on a correctly functioning unreplicated node take t units of time to compute the response to an input message. The corresponding correct output from a fail-silent node will take at most $t'=t+t_{delay}$ units of time, where t_{delay}, $t_{delay}>0$, is the bounded worst-case delay introduced by the redundancy management protocols. If the output from the fail-silent node is produced later than t' then the node will be said to have suffered a performance failure [4]. A fail-silent node can be in one of the three states (see fig. 2).

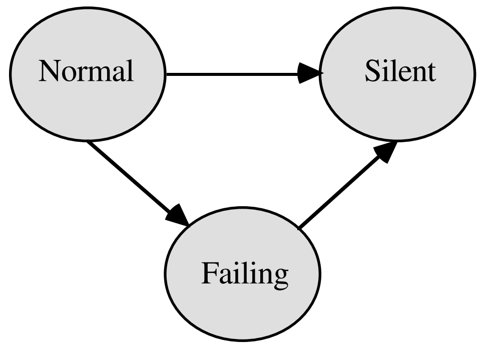

Fig. 2: Fail-silent node states

(i) Normal State: In this state, a node produces correct outputs. Detection of an internal failure (by the comparator process) causes the node to irreversibly enter either the *failing* state or the *silent* state.

(ii) Failing State: This is an intermediate state in which the node can suffer at most one performance failure. From this state the node eventually enters the terminal silent state.

(iii) Silent State: No new valid messages are produced by the node. Any messages produced by the node can only be invalid or copies of previously produced valid messages: any functioning destination node can detect these messages as unwanted.

The reason for the existence of the intermediate failing state is as follows. A faulty processor can contain a message from the correct processor sent for comparison (a message that was sent before the correct processor stopped). The faulty processor can output this as a valid double signed message at any future time. The Sender and Comparator processes of each processor must therefore incorporate intra-node message synchronisation measures to ensure that each processor of a node at any time has no more than one message which has been sent to the neighbour for comparison but has not yet been compared locally; in this way, the number of performance failures in the failing state can be limited to at most one.

The fact that a fail-silent node can suffer a single performance failure in the intermediate state need not be a cause for concern in most applications. Consider a system of "fail-crash" nodes without an intermediate state. A client application with timing constraints and expecting a response from such a node would still be expected to contain timeliness checks for detecting an absent response. The same checks will be adequate for the case of fail-silent nodes for filtering out late responses. If application programs have no timing constraints, then a performance failure suffered by a fail-silent node in the failing state will not cause any inconsistencies.

Thus, a system of software implemented fail-silent nodes can be regarded as capable of implementing the abstraction of fail-silence in the following sense: a node produces either correct messages which can be verified as such by destination fail-silent nodes, or it ceases to produce new correct messages, in which case destination nodes can detect any messages it may produce as unwanted.

It is possible to design specialised fault-tolerant network interfaces that could prevent further messages from being output by a node once one of the processors detects a failure. Minimally, we need to provide a network interface with a single switch that can unilaterally and irreversibly be switched off by a control signal sent by either of the processors in the node.

Any software solution to the design of a node that has no intermediate failing state will require additional redundancy. For example, one could delegate the responsibility of message comparison and output to a separate node that does not fail. A $2\pi+1$ failure-masking node (capable of masking up to π processor failures within a node) could provide the services of message comparison and output to a collection of $\pi+1$ processor nodes. Indeed the failure-masking node can provide other services, such as recording the status of fail-silent nodes. This design very much resembles that of a system of fail-stop nodes [19] that can switch from the functioning to the halted state, and can provide failure-status indication.

3.4. Rationale behind the Experimental Work

In the rest of the paper we will be describing our experimental work on evaluating a number of designs for two-processor fail-silent nodes. However, before that, a brief discussion on the rationale behind our experimental work is worth a mention. We note that the performance of a fail-silent node will depend on how quickly messages can be ordered and compared. Ordering can be achieved in several ways. The basic idea is to have an agreement protocol which guarantees that all correct replicas receive the same set of messages and then accomplish ordering by assigning monotonically increasing sequence numbers to messages. It is also necessary to devise a method to establish when a message becomes *stable*, i.e. when it is guaranteed that no valid messages with sequence numbers less than a certain value, *seq*, will ever be received, so that all messages with sequence numbers less than *seq* can be processed in a consistent order among all the replicas. General methods for assigning sequence numbers to messages, and associated stability tests for different system assumptions have been discussed in [20]. We have used these ideas and applied them to the special case of two-processor fail-silent nodes. The delay imposed by the comparison protocol will mostly be made up of the time spent in message exchanges plus any delay introduced by the intra-node message synchronisation measure necessary to ensure that each processor of a node at any time contains no more than one message from the neighbour for comparison.

We took the following approach in our quest for a design that minimised both ordering and comparison delays. First we performed a reference implementation based on a design that was relatively easy to understand. For this reason, in the reference implementation we used a simple order protocol for messages and a simple comparison protocol that did not incorporate any synchronisation measure for limiting the number of received messages from the neighbour to just one (potentially, such a node can suffer more than one performance failure in the failing state). We then investigated a number of ways of reducing message ordering delays. After this we investigated message comparison protocols with synchronisation measures. Our work on order protocols proved highly significant in coming up with a clean and efficient solution. Having selected a design for the comparison protocol, we undertook comparative performance evaluation of four node designs, all using this comparison protocol but with different order protocols for input messages, starting with the one used in the reference implementation. We had carefully designed the software of the reference implementation in a modular fashion; this made it relatively easy for us to replace or modify modules to incorporate the necessary changes [24].

4. REFERENCE IMPLEMENTATION

4.1. Software Architecture

The overall software architecture of a fail-silent node is depicted in fig. 3, where the major software modules within a processor of a node and their interactions are summarised. A processor maintains several message queues and lists:

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

b) Delivered Message Queue_i (DMQ_i): Contains ordered messages to be consumed by the application process Service_i.

c) Processed Message Queue (PMQ): Contains unsigned output messages produced by local application processes. These messages must be *validated* by the Comparator process before transmission to the final destination. So, the Sender process is responsible for transmitting messages in PMQ to the neighbour processor, as well as to the local Comparator process.

d) External Candidate Message List (ECL): Contains singly signed messages that have been received from the neighbour processor for validation.

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

f) Compared Message Queue (CMQ): Contains successfully compared and double signed messages (valid messages) ready to be transmitted over the network.

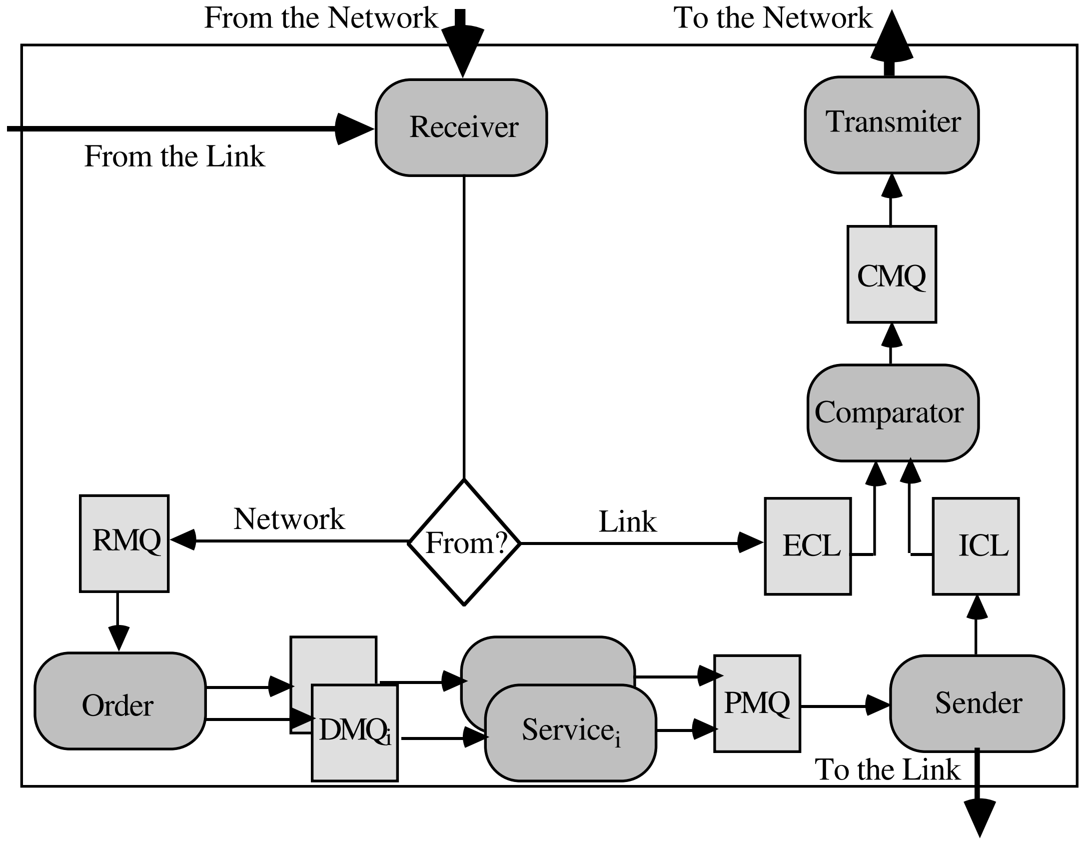

Fig. 3: Software architecture of a processor in a node

4.2. Comparison protocol

The reference implementation uses a very a simple comparison protocol: referring to fig. 3, the Sender process of a processor transmits messages from the PMQ to the neighbour, where they get buffered in the neighbour's message pool ECL. The Comparator process maintains, for each application process Service_i, the sequence number of the next message to compare (recall that application processes assign monotonically increasing sequence numbers to new messages they produce). Using this criterion, the Comparator matches messages with identical sequence numbers from ECL and ICL; a comparison that detects a disagreement indicates a failure. Similarly, an absence of a message for comparison (after a node specific time-out interval) also indicates a failure. Once a failure is detected, the comparator process stops, and so does the sender process.

In this simple protocol, the ECL of a processor is permitted to contain more than one correct message from the neighbour; thus potentially, a faulty processor can output more than one late valid message. In a latter section we will describe the additional synchronisation measure necessary to prevent this from happening.

4.3. Order Protocol with Synchronised Clocks

Our reference implementation of the order protocol, to be described in this section, makes use of the well-known approach of using synchronised clocks for message ordering. The clocks of both processors in the node are assumed to be synchronised such that the magnitude of the measurable difference between readings of clocks at any instant is bounded by a known constant, say ε. Because the non-faulty processor stops as soon as a failure is detected, the clock synchronisation protocol need not be fault-tolerant, and can be assumed to execute in a fault-free environment. It has been shown that the lower bound on ε is $\delta/2$ [5]; so in a fault-free environment, ε can be taken as $\delta/2$ provided the inter-synchronisation period is kept small enough so that the effects due to differences in the running rates of clocks can be ignored. The Order process of a processor timestamps a message to be ordered with its local clock reading. A copy of the timestamped message is 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 $T+\Delta$, where $\Delta=\delta+\varepsilon$. Once a message with timestamp T becomes stable, no valid messages with timestamp $T'<T$ can be received by an Order process. Stable messages are enqueued in the appropriate DMQ_i in increasing timestamp order (with the action being taken to discard, rather than to enqueue a stable message, if its replica has already been enqueued).

The Order process is composed of three cyclic processes: *Relayer, Transfer* and *Deliver* (see fig. 4). 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). The Transfer process receives relayed messages from the link, and performs a timeliness check that rejects any message received too early (messages with timestamp less than C-ε, where C is the current reading of the processor's clock) or received too late (messages with timestamp greater than $C+\Delta$). Accepted messages are inserted into the OML. The Deliver process takes stable messages (messages with timestamp less than C-Δ) from the OML, removes duplicates and enqueues the messages on the appropriate DMQ_is in increasing order of timestamps.

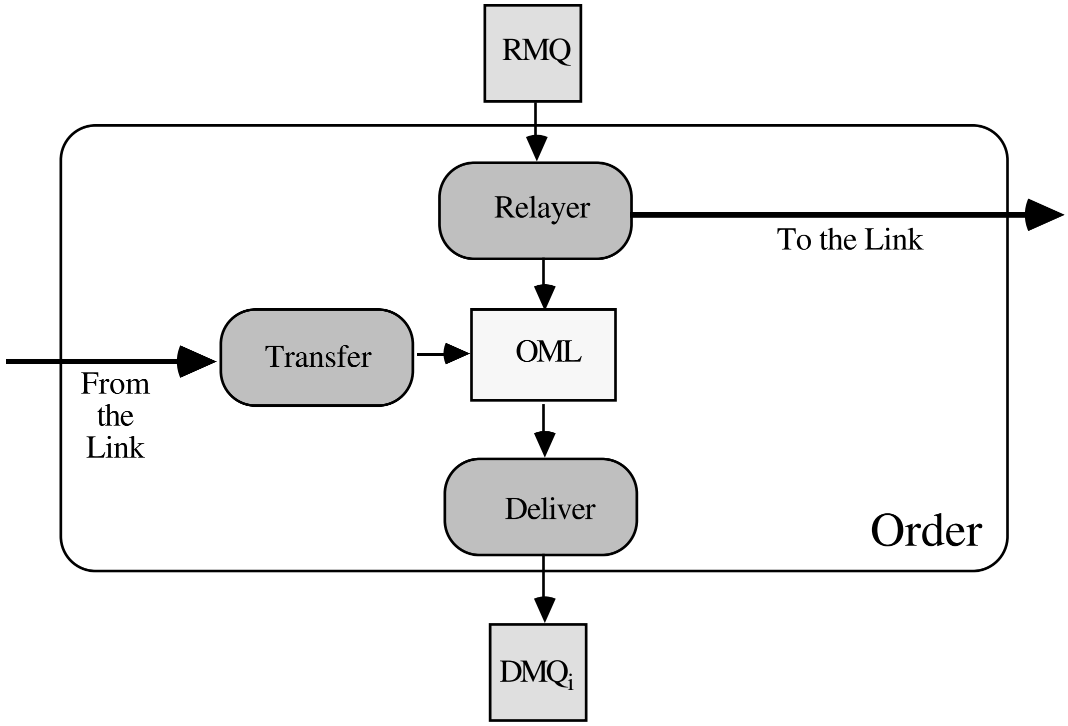

Fig. 4: Order protocol with synchronised clocks

To compare the ordering speeds of various protocols in failure-free situations, we will define the *actual stability delay* (Σ_a) for an order protocol in terms of a reliable *reference* clock. (Such a clock could be a correct processor's physical clock.) When both the processors of a node are correct, Σ_a of an order protocol for a given message from the network is defined as the reference clock time that elapsed between the instant a copy of the message is first received by one of the processors of the node and the instant that message gets ordered and enqueued in the appropriate DMQ$_i$s of both the processors in the node. Throughout this paper, we will assume that the effects of differences in the runnning rates between the reference clock and any correct processor's clock are negligible when intervals such as ε, δ and Δ are measured. With this assumption, Σ_a of the order protocol just presented will be:

$$\Sigma_a = \Delta + \min\{\alpha\varepsilon_a, \lambda_a\};$$

where λ_a ($\lambda_a \geq 0$) is the magnitude of the message reception skew according to the reference clock, i.e. the difference between the reference clock times when each processor in the node receives a copy of the message from the network, ε_a ($0 \leq \varepsilon_a \leq \varepsilon$) is the magnitude of the actual clock synchronisation error at the time the message is first received from the network, and α is the *ahead* factor which is 1 if the clock of the processor that first received the message from the network is ahead of the other processor's clock, or zero if either the first processor's clock is not ahead or λ_a=0. Note that if only one processor receives the message from the network and the other does not, then $\lambda_a=\infty$, but the message will be ordered at both the processors.

We will also define Σ_{min} and Σ_{max} to be respectively the lower, and the upper bound of the actual stability delay of an order protocol ($\Sigma_{min}\leq\Sigma_a\leq\Sigma_{max}$). Therefore, for the above protocol we have:

$$\Sigma_{min}=\Delta;\ \Sigma_{max}=\Delta+\varepsilon;\ \text{and}\ \Delta\leq\Sigma_a\leq\Delta+\varepsilon.$$

The fixed overhead of at least Δ units of time implicit in this order protocol has motivated us to seek enhancements. We begin by describing a method for improving the above protocol and then describe new protocols that do not require the clocks of a node to be kept synchronised.

5. IMPROVED ORDER PROTOCOLS

5.1. Improving the Synchronised Clock Algorithm

The arrival of a relayed message can be used to reduce the constant stability delay Δ imposed by the order protocol. We shall assume that messages sent over the link are received in the sent order. Given this *fifo* assumption, the timestamp of a received relayed message can be used to define a new lower bound on the actual stability delay. Fig. 5 will be used to illustrate the idea.

In case (a) a relayed message with timestamp T is received and the local clock reading, C, is greater than T. As no more messages will be received for ordering from the neighbour bearing a timestamp smaller than or equal to T, and any new local message for ordering will get a timestamp greater than or equal to C, all messages from that sender for ordering (in OML, fig. 4) with timestamps smaller than or equal to T are stable.

Case (b) shows the case where a message with timestamp T is received for ordering from the neighbour and $C<T$. In this case all messages for ordering with timestamp smaller than C are stable. Note that in this case it is guaranteed that the neighbour's clock is ahead of the processor's clock, and also that the message could not have taken more than $\delta/2$ time in transmission across the link. (Otherwise, it is not possible to have $C<T$ with ε being $\delta/2$.) Therefore, updating the local clock to $T+1$ will not cause the magnitude of the clock difference to increase beyond $\delta/2$, i.e., beyond ε.) With this update, a relayed message with timestamp T received by a processor will define a new stabilisation interval such that all the messages with timestamp smaller than or equal to T are stable (case (c)). In other words, any message relayed from one processor to the other becomes stable at the receiving processor as soon as it is received,

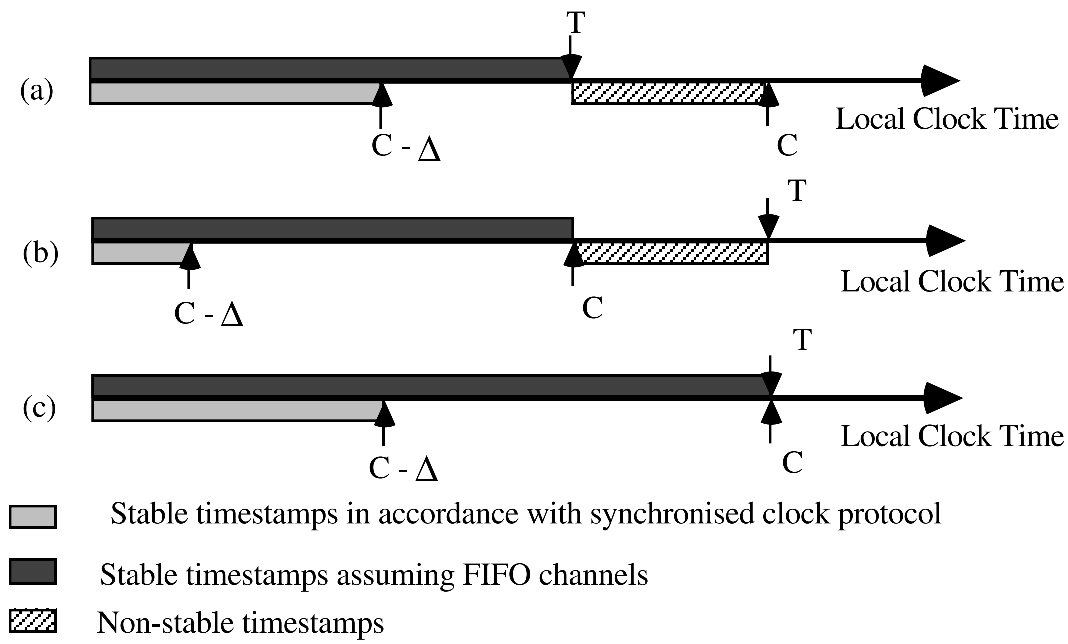

Fig. 5: Stability intervals

To derive Σ_a for the modified protocol, let the processor that is first to receive a message from the network receive it at reference clock time T_r. The other processor will receive the relayed message at time $T_r+\delta_a$ (where δ_a, $0\leq\delta_a\leq\delta$, is the actual link transmission delay) and can immediately order it. The first processor will be able to order the message at time $T_r+\Delta$ or at time $T_r+\lambda_a+\delta_a$ if it receives the relayed message from the other processor before $T_r+\Delta$. So, we have:

$\Sigma_a=\min\{\Delta, \lambda_a+\delta_a\}$; and, $\Sigma_{min}=0$; $\Sigma_{max}=\Delta$; and $0\leq\Sigma_a\leq\Delta$.

5.2. Order Protocol with Logical Clocks

We can take the idea discussed before a step further and eliminate the requirement of having the physical clocks of the processors forming a node to be kept synchronised, and instead use logical clocks for generating timestamps [14].

In this order protocol each processor of a node maintains two logical clocks (counters), namely the local logical clock (*LLC*) and the remote logical clock (*RLC*), which are initialised to 1, and 0 respectively. *LLC* is used to timestamp messages relayed to the neighbour for ordering, while *RLC* is used to store an "estimation" of the neighbour's *LLC*. These clocks are updated in the following way: whenever a processor relays a message to its neighbour, it timestamps the message with the current value of *LLC* , and increments *LLC* by one; whenever a message with timestamp *T* is received from the neighbour, *RLC* is set to *T* and *LLC* is set to the maximum of its current value and *T*+1. These updates ensure the following properties:

(i) messages are relayed to the neighbour bearing increasing timestamps; and

(ii) the value of *RLC* of a processor is smaller than that of the *LLC* as well as that of its neighbour's *LLC*.

Property (ii) guarantees that all messages for ordering with timestamps smaller than or equal to *RLC* are stable. So, as before, a relayed message becomes stable at the receiver processor as soon as it is received, and the actual stability delay will be:

$$\Sigma_a=\lambda_a+\delta_a.$$

The protocol as presented above has one shortcoming. Messages at a processor can become stable only after the arrival of a relayed message from the neighbour (because *RLC* is updated only when a message relayed from the neighbour is received). However, a processor can only relay a message if it receives it from the network, so if only one of the processors receives a message from the network ($\lambda_a=\infty$), it will be prevented from stabilising that message. To solve this problem we discuss a scheme based on time-outs that allows a processor to update *RLC* even if the other processor does not relay a message [20].

When a processor (say P_1) relays a message (say m_1) with timestamp T to its neighbour (say P_2), it schedules an update of *RLC* to value T to occur at time $t+2\delta$, where t is the value read on its physical local clock when m_1 was relayed. At time $t+2\delta$, *RLC* is updated to T only if its value is less than T. The 2δ time-out interval follows from the fact that after receiving m_1 with timestamp T, *LLC* of P_2 will have the value of at least $T+1$; therefore any message with timestamp smaller than or equal to T relayed from P_2 to P_1 (say m_2) will have been relayed before P_2 received m_1. In the worst case, this would be done just before the reception of m_1, with m_1 and m_2 each taking δ units of time. Thus P_1 must wait for at least 2δ units of time before advancing its *RLC*.

The Order process of this protocol is also composed of the three cyclic processes which work in a fashion similar to those discussed in the previous protocol (see fig. 4). The Relayer process picks up a message on its RMQ, timestamps it with the value T read on *LLC*, and places the message on its OML. Then, a copy of the timestamped message is sent over the link to the neighbour processor. Finally, the processor's *LLC* is incremented by one, and an update of *RLC* to T is scheduled to be executed in 2δ units of time. The Transfer process receives a relayed message with timestamp T from the link, performs a timeliness check (a message is considered timely if its timestamp is greater than the current value of *RLC*), and if timely, places it in the processor's OML. *LLC* and *RLC* are then updated if necessary as discussed before. Messages in OML with timestamps less than or equal to *RLC* are stable. Thus we deduce:

$$\Sigma_a=\min\{2\delta, \lambda_a+\delta_a\};\ \Sigma_{min}=0;\ \Sigma_{max}=2\delta;\ \text{and } 0\le\Sigma_a\le 2\delta.$$

5.3. Asymmetric Order Protocol

We now present a protocol where we assign different roles to each of the two processors forming a node. We will term one processor the *leader* and its neighbour the *follower*. 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 inspiration for this way of building a fail-silent node comes from the leader-follower replication protocol for application level processes used in the Delta-4 system [1, 16]). Due to the simplicity of this ordering mechanism, there is no need for a special Order process within a processor. Instead we will have Receiver processes with different functionality in the leader and in the follower.

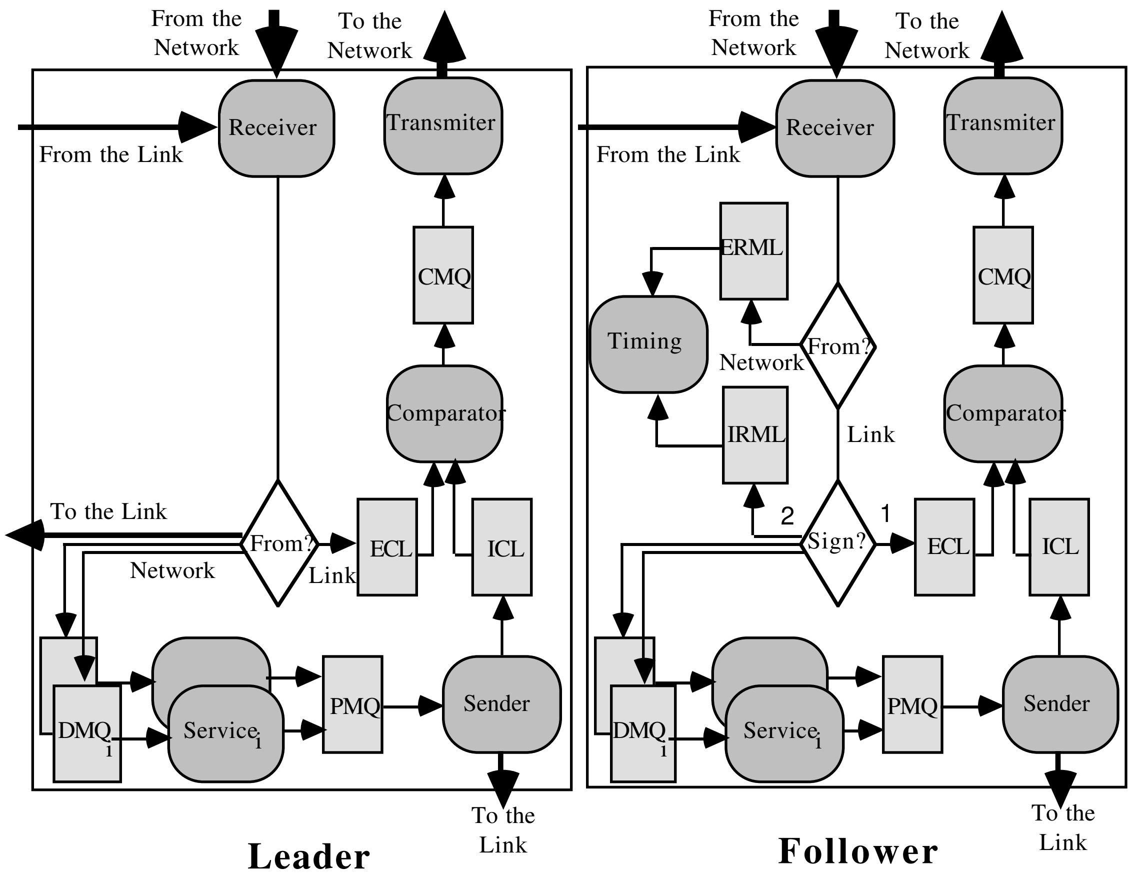

Fig. 6: Leader-follower fail-silent node

The node works as follows (see fig. 6); the leader maintains a counter whose value is used for assigning unique identifiers to input messages. An authentic double signed message received by the Receiver of the leader is tagged with the counter's value, and the counter is incremented by one. The message is then deposited in the appropriate DMQ_i in increasing order of tag values and a copy of the message is also sent to the follower across the link. Output messages from an application process, $Service_i$, follow the same path as discussed before. Tagged messages from the leader reach the follower where they also get deposited in the appropriate DMQ_is. Message buffers ECL, ICL, CMQ and the comparator process have the same role as before.

The asymmetry introduced by assigning different roles to the two processors of a node requires us to introduce an extra mechanism in the follower for detecting late or non arrival of a message for ordering from the leader. A Timing process (see fig. 6) is introduced in the follower. The follower's Receiver process deposits each authentic double signed input message received from the network in the External Received Message List (ERML) with an associated time-out τ. Copies of messages received from the leader via the link and on their way to DMQ_i, are deposited in the Internal Received Message List (IRML). The Timing process picks up each message in the IRML and resets the time-out associated with its counterpart (if any) in the ERML. If a time-out expires, the follower assumes that the leader has failed to send a message for ordering, and stops the activities of all the processes in its processor.

Unlike the previous protocols, in order to calculate the actual stability delay of this protocol it is relevant to identify the processor that first receives a copy of a particular input message. We will define λ_{LF} as the difference between the time that the leader receives a copy of a particular input message, and the time that the follower receives a copy of the same message. The actual stability delay for this protocol is then given by:

$$\Sigma_a=\Sigma_F=\Sigma_L+\delta_a, \text{ and } \Sigma_L=\begin{cases}0, \text{ if } \lambda_{LF}<0\\ \lambda_{LF}, \text{ otherwise;}\end{cases}$$

where Σ_L and Σ_F are the actual stability delay for leader and follower, respectively.

The above protocol can be embellished to deal with the case where a correctly functioning leader does not receive a message from the network, but the follower does, which leads to the node becoming silent. The follower processor can try to prevent a shut down by feeding the leader with the missing input message. In this 'feedback' version of the leader-follower protocol, after a time-out τ has expired, the follower sends a copy of the missing input message to the leader in order to have it properly ordered (for simplicity, this path is not shown in fig. 6). A second time-out τ', $\tau'\geq 2\delta$, is associated with the message. If this time-out also expires, then the follower may assume that the leader has failed, and the follower will cease its own activities. The stability delays become:

$$\Sigma_a = \Sigma_F = \Sigma_L + \delta_a, \text{ and } \Sigma_L = \begin{cases} 0, \text{ if } \lambda_{LF} < 0 \\ \min\{\lambda_{LF}, \tau+\delta_a\}, \text{ otherwise.} \end{cases}$$

Also,

$$\Sigma_{min}=0,\ \Sigma_{max}=\tau+\tau', \text{ and } 0 \le \Sigma_a \le \tau+\tau'.$$

A sensible strategy is for the follower to set $\tau=0$ (thus, as soon as the follower receives a message from the network, it checks for the presence of the corresponding relayed message from the leader) and $\tau' = 2\delta$, thus Σ_{max} for this protocol becomes identical to the logical clock protocol.

6. ASYMMETRIC COMPARISON PROTOCOL

The comparator protocol discussed before permitted a node in the failing state to commit more than one performance failure. One way of preventing this from happening is to use a comparison protocol that guarantees that a processor sends a given message for comparison to its neighbour *only after* all previously sent messages have been successfully compared locally. In order to prevent deadlocks, it is also necessary that the processors first agree on the order in which they have to exchange messages for comparison. In our architecture, a logical way of achieving this agreement would be to insert an order process between the PMQ and the Sender process of each processor. The asymmetric ordering approach discussed in the previous section provides a very convenient way of integrating ordering with comparison. Accordingly, we present a comparison protocol based around the leader-follower technique. It is worth noting that our comparison protocol can be used within a node that uses any order protocol for input messages (synchronised clock, logical clock or the leader-follower); this is because ordering for input messages is independent from ordering for output messages. The description to be given here concentrates on the message synchronisation aspects of the protocol, the other aspects remain unchanged.

For the purpose of message comparison then, one processor is assigned the role of a leader, and the other, the follower. In the leader, the messages in the PMQ follow the same path as before (see fig. 3). However, the following synchronisation between the Sender and the Comparator is introduced: the Sender is allowed to send a new message over the link for comparison only if permitted by the Comparator, and this permission is granted by the Comparator after it has finished comparing the current message.

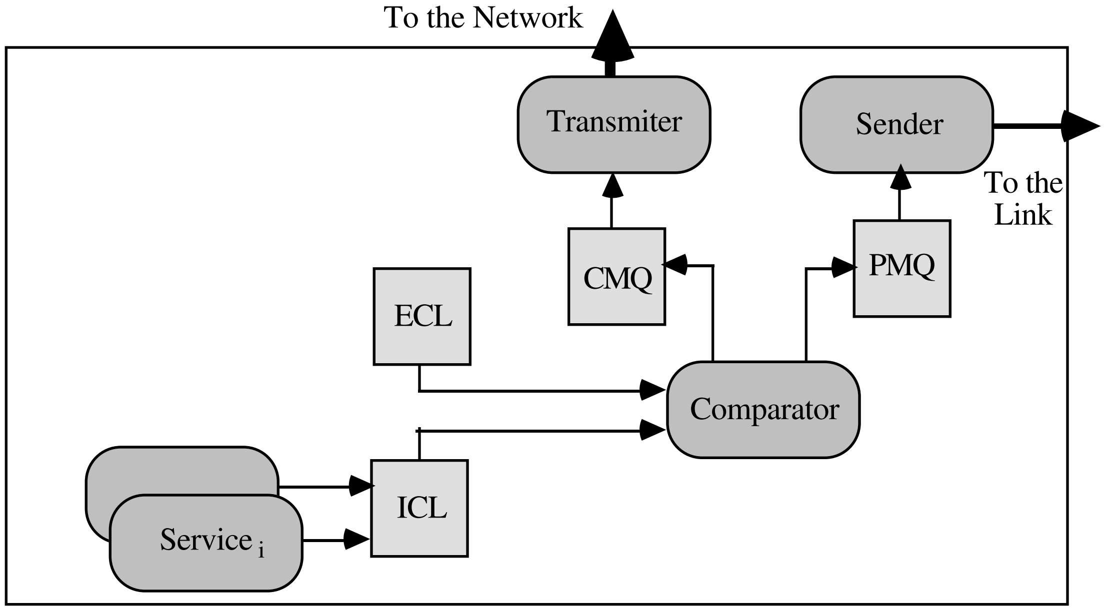

Fig. 7: Message comparison

On the follower's side, messages produced by application processes follow a slightly different path, as shown in fig. 7. The Comparator compares the message in the ECL (sent by the leader) with the locally produced one in the ICL; if the comparison succeeds, the valid message is deposited in the CMQ for network delivery and the locally produced message is deposited in the PMQ for delivery over the link to the leader. This message will arrive in the ECL of the leader, get compared and, if successful, the Comparator process of the leader will then permit the local Sender process to send the next message for comparison.

7. PERFORMANCE EVALUATION

In this section we present the performance figures obtained after a set of experiments we have run. Our main objective has been to assess the degradation in performance suffered by a node as it is called upon to execute the redundancy management software not present in an ordinary processor. Currently, simple checksums are being used as signatures and so have a minimal impact upon system performance. The need for more complex signature mechanisms has not yet been assessed.

We have implemented fail-silent nodes on T800 Inmos transputers and evaluated their performance under four protocols for ordering input messages: (i) the reference implementation based on clock synchronisation algorithm; (ii) logical clocks; (iii) leader-follower; and (iv) leader-follower with feedback. All these implementations made use of the asymmetric message comparison protocol discussed before (for the cases (iii) and (iv), the processor acting as the leader for ordering was also the leader for comparison). The two processors of a node are directly connected to each other by transputer links, thereby providing a fast internal path for intra-node communication.

The first experiment consists of a client application process executing on a node, and requesting a simple service from a server application process which executes on a different node. The client process issues a request to the server process and waits for the response. The server process receives a request from the client, services it (the actual computation performed is minimal) and sends the response back to the client. Upon reception of the response message the client issues a new request. We have measured the following time intervals for the server process:

(i) *Input delay (ID)*: The Input delay measures the time interval between a message entering the node (the earliest of the reception times at the processors) and the message being last removed from DMQ_i by one of the processors. The delay is made up of the actual stability delay for a message (Σ_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.

(ii) *Output delay (OD)*: The Output delay measures the time interval between a message becoming ready for comparison at both the processors (i.e. largest of the two times the message is entered in the PMQ) and the message being output by the node (i.e. being first output by one of the processors). It reflects the time taken for a message to be compared, and output.

(iii) *Node delay (ND)*: Finally, 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.

Model/Delays(ms)	ID	OD	ND	δ_{av}	λ_{av}
Synchronised Clocks	20.21	4.09	24.30	3.47	1.44
Logical Clocks	7.64	3.18	10.82	3.94	1.50
Leader-Follower	4.34	2.06	6.40	2.32	1.23
Leader-Follower (feedback)	4.79	2.48	7.27	3.07	0.89

Table I. Performance figures for a client-server system

We have collected data for ten runs of experiments; each run involves the client node sending 100 request messages of 64 bytes. For each one of the time intervals discussed above we have averaged the values measured for each of the requests processed. We have also measured the average link transmission delay (δ_{av}), and the average message reception skew (λ_{av}). The average delays obtained are summarised in Table I, where the figures are expressed in milliseconds.

(i) Unreplicated Node: We have also executed the experiment using single processor nodes. 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 server amounted to about 1ms, of which about 0.7ms was due to input overheads, whilst about 0.3ms was due to output overheads.

(ii) Nodes with synchronised clock order protocol: Experiments under worst case circumstances determined the smallest safe value for δ to be 12ms. This reference implementation of a node uses a simplified version of the clock synchronisation algorithm presented in [9]. As stated before, ε can be set to $\delta/2$, hence we fixed ε=6ms which gives the stability delay, Δ, of 18ms (since $\Delta=\delta+\varepsilon$). Measurements indicated that the actual stability delay is almost the same as Δ, so the values shown in Table I for ID indicate that the overheads due to message authentication and queue manipulation take up to 2.21ms.

(iii) Nodes with logical clock order protocol: Using logical clocks, the actual stability delay would be around $\delta_{av}+\lambda_{av}$. Assuming the overheads due to message authentication and queue manipulation to be same as above, the results given in Table I show that this expectation of Σ_a is almost realised in practice. Unlike the previous protocol, this and the asymmetric protocols have their performance proportional to the actual values of transmission delays and message reception skews.

(iv) Nodes with leader-follower order protocols: For the asymmetric order protocols, 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 previous section, ID corresponds to the follower's stability delay ($\Sigma_a=\Sigma_F=\Sigma_L+\delta_a$), plus any overhead due to message authentication and queue manipulation. In our experiment, the two nodes were directly connected by leader-to-leader and follower-to-follower transputer links. Therefore, because the follower always outputs messages before the leader, most of the time it will also be the follower who will receive a copy of a particular input message first. Thus, most of the time we will have $\lambda_{LF}>0$, and consequently $\Sigma_a=\lambda_{LF}+\delta_a$. The values shown in Table I indicate that the message handling overheads for the asymmetric protocols (0.79ms for the leader-follower, and 0.83ms for the leader-follower with feedback) are close to those experienced by the unreplicated node. This is because the functions of the order protocol are incorporated into the Receiver process (the overheads are slightly bigger because in the replicated node messages must be authenticated). From the performance figures presented for the two leader-follower protocols, we see that the extra message traffic introduced by the feedback mechanism has hardly any impact on the performance of the node.

Despite the fact that all the implementations make use of the same comparison protocol, figures in Table I show that a node with an asymmetric order protocol for input messages suffers less output delay than the node with the symmetric one. The reason for this is that the asymmetry introduced for input ordering and for comparison helps the follower at comparison time: by the time a message becomes available in the ICL (see fig. 7), the leader's message will usually be available the ECL.

Our next experiment was performed to evaluate the impact of the size of input messages (messages that need to be ordered) on the performance of a node. The size of messages will affect intra-node message transmission times, consequently affecting both input and output delays. Transputers use a byte-stream protocol for link-level communication. On our system, the end-to-end message transmission delay between two transputers varied from 1.8ms (messages of size 256 bytes) to 3.3ms (messages of size 2048 bytes). Using the same client-server system, we measured the node delay for the various order protocols as the message size was increased from 256 to 2048 bytes (see fig. 8).

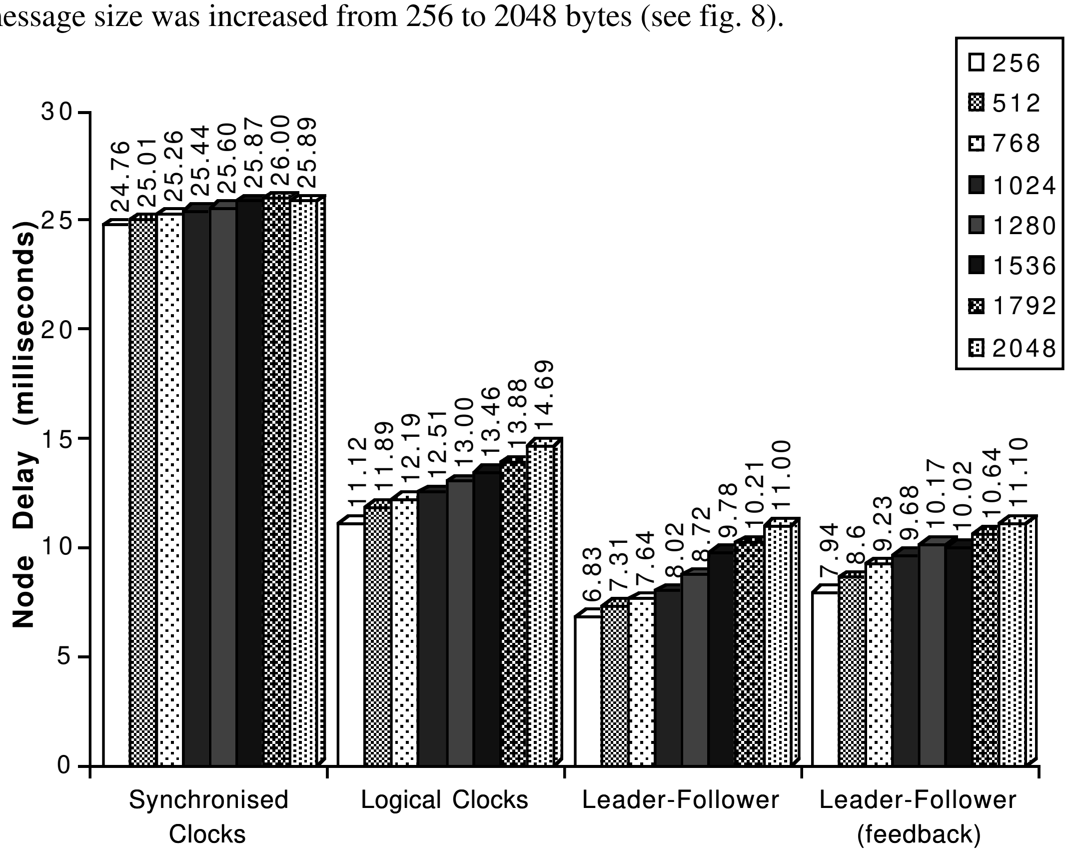

Fig. 8: Impact of message size

The impact of message size on order protocols will not be uniform. The increased transmission delay will have little impact on the performance of the order protocol based on synchronised clocks, because its stability delay is based on the worst case transmission delay. Thus the node delay for the synchronised clock protocol suffers a moderately small increase of 1.24ms (from 24.76ms to 26.00ms), mainly due to the increased output delay. On the other hand, as we would expect, other protocols would be affected more strongly: the values in fig. 8 show an increase of 3.57ms for the logical clock protocol and increases of 4.17ms, and 3.16ms for the leader-follower, and the leader-follower with feedback protocols, respectively.

In our last experiment we measured the maximum throughput: the maximum rate a node with a given order protocol can order and compare messages. We have compared the throughput of each node configuration with the throughput of the unreplicated node. For this experiment we have used a fixed message size of 64 bytes and a modified version of the client process. The client process now does not wait for the response to arrive before issuing the next request; rather it sends a continuous stream of request messages. The experiment simulates the environment where a server process always has input messages for processing. We have measured the rate (messages per second) at which messages were deposited in the CMQ by the comparator of the processor that first output a message (see fig. 3). This output rate (OR) was then used to obtain the throughput ratio TR: $(OR/OR_{unreplicated})$ where $OR_{unreplicated}$ is the output rate measured for the unreplicated node. The figures obtained are presented in table II.

Model	OR (msg/sec)	TR (%)
Unreplicated node	329	100.00
Synchronised Clocks	66	20.06
Logical Clocks	68	20.67
Leader-Follower	130	39.51
Leader-Follower (feedback)	111	33.74

Table II: Throughput of a heavily loaded node

Under a heavy load, the ordering protocols will have their performance closer to the worst case. We see that the performance of the node with logical clock protocol is almost the same as the synchronised clock based node. The asymmetric protocols still out-perform the other protocols.

8. CONCLUDING REMARKS

We have described our work on building efficient fail-silent nodes. We first performed a reference implementation that made use of a simple comparison and order protocols. We have then investigated how the performance of the order protocol can be improved; this led to a much simpler protocol based purely on logical clocks, obviating any need for keeping intra-node clocks explicitly synchronised. We have also designed and implemented asymmetric order protocols. We then described how the asymmetric ordering approach can also be exploited for the construction of an efficient message comparison protocol. Extensive experiments were performed to evaluate the performance of nodes under these order protocols. The results obtained indicate that adopting the asymmetric leader-follower mechanism within a fail-silent node for message comparison as well as for ordering represents the best design choice. It must be stated here that it is possible to design a symmetric comparison protocol that does not require processors to decide order for exchanging messages for comparison. In such a protocol, the Sender and Comparator processes of a processor ensure that at any given time there is no more than one message that has been sent for comparison before being locally compared first. Combining this protocol with other symmetric ordering protocols discussed earlier could result in other efficient node designs.

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 obtained through improved message passing mechanisms, so the leader-follower node described here probably indicates the limits of what can be achieved using standard 'off-the-shelf' processors and asymmetric protocols. In our particular implementation, the performance impact of using fail-silent nodes is to produce a delay in response of about 6ms per message in a lightly loaded system. Secondly, under worst case loading, a fail-silent node can achieve about 39% throughput rate of its non-replicated counterpart. It should be appreciated that this price in performance becomes significant in only those distributed applications where processes interact frequently. If on the other hand, application processes are involved in computations requiring little interactions then the performance impact of adding software-implemented fail-silence can be quite small. Thus, bearing in mind the discussion presented at the start of the paper on the advantages of software-implemented fail-silent nodes over hardware-implemented nodes, we can anticipate a range of applications for which these software-implemented nodes offer an attractive alternative to their hardware-implemented counterparts. We conclude by highlighting some of our recent work that further illustrates the advantages of the software-implemented approach.

The software approach makes it possible to apply the fail-silence measures *selectively*, only to those processes that are deemed critical in a given application. The Voltan system software that uses the asymmetric leader-follower mechanism is sufficiently lean to make it practical to use it as a software library for constructing *self-checking process-pairs*. Each member of a process-pair contains a number of threads that together implement the entire Voltan message ordering and self-checking mechanisms. We have implemented the system software that permits a collection of distributed processes to be replicated transparently giving an equivalent collection of self-checking Voltan processes [3].

The software approach also makes it possible to extend the capabilities of a node with relative ease. We have proposed a simple, but significant embellishment to the capabilty of a fail-silent node; the resulting node has been termed a *fail-stable* node [8]. In addition to the fail-silence property, a π+1 processor fail-stable node has the second property of providing a *stable store*: the node maintains a log whose contents survive any internal failure. The log is accessible to other nodes in the system, and can be used for constructing the most recent states of processes running on the node before the node stopped. The state information provided by a halted node facilitates easy and prompt restarting of the stopped processes on other nodes. Such a node therfore forms an attractive building block for constructing available distributed systems.

ACKNOWLEDGEMENTS

This work has been supported in part by grants from the UK Engineering and Physical Sciences Research Council and the Brazilian Research Council (CNPq).

REFERENCES

[1] P.A. Barrett et al, "The Delta-4 Extra Performance Architecture (XPA)," Digest of Papers, FTCS-20, Newcastle upon Tyne, pp.481-488, June 1990.

[2] P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", IEEE Computer, Vol. 21(2), pp. 37-45, February 1988.

[3] D. Black, C. Low and S.K. Shrivastava, “The Voltan Application Programming Environment for Fail-Silent Processes”, Tech. Report, Dept. of Computing Science, University of Newcastle upon Tyne, January 1996.

[4] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," Communications of the ACM, Vol. 34(2), pp. 57-78, February 1991.

[5] D. Dolev, J. Halpern and H.R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronization," Proceedings of the 16th ACM STOC, Washington, D.C., pp. 504-511, May 1984.

[6] C-J. L. van Driel, et al, "The Error-Resistant Interactively Consistent Architecture (ERICA)," Digest of Papers, FTCS-20, Newcastle upon Tyne, pp. 474-480, June 1990.
[7] P.D. Ezhilchelvan and S.K. Shrivastava, "A Distributed Systems Architecture Supporting High Availability and Reliability," Dependable Computing and Fault-tolerant Systems, Vol. 6 (ed. J.F. Meyer, R.D. Schlichting), pp. 67-91, Springer-Verlag, 1992.
[8] P.D. Ezhilchelvan, C. Low and S.K. Shrivastava, “Building Available Distributed Systems using Fail-Stable Nodes”, Tech. Report, Dept. of Computing Science, University of Newcastle upon Tyne, January 1996.
[9] J.Y. Halpern, B. Simons, H.R. Strong and D. Dolev, "Fault Tolerant Clock Synchronization," Proceedings of the Third ACM Symposium on PODC, Vancouver, pp. 89-102, August 1984.
[10] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," IEEE Transactions on Computers, Vol. 37(4), pp. 398-405, April 1988.
[11] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," Digest of Papers, FTCS-20, Newcastle upon Tyne, pp. 466-473, June 1990.
[12] J.H. Lala, "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications," Digest of papers, FTCS-16, Vienna, Austria, pp. 338-343, July 1986.
[13] J.H. Lala, and L.S. Alger, "Hardware and Software Fault Tolerance: A Unified Architectural Approach," Digest of Papers, FTCS-18, Tokyo, Japan, June 1988, pp. 240-245.
[14] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21(7), pp. 558-565, July 1978.
[15] D.L. Palumbo, and R.W. Butler "Measurements of SIFT Operating System Overhead," NASA Tech. Memo. 86322, 1985.
[16] D. Powell (Ed.), Delta-4 - A Generic Architecture for Dependable Distributed Computing, Spring-Verlag, 1992, ISBNs 3-540-54985-4, and 0-387-54985-4.
[17] J. Reisinger, and A. Steininger, "The Design of a Fail-Silent Processing Node for the Predictable Hard Real-Time System MARS," Distributed System Engineering Journal, Vol. 1(2), pp. 104-111, 1993.
[18] R. Rivest, A. Shamir and L. Adleman, "A Method of Obtaining Digital Signatures and Public-key Cryptosystems," Communications of the ACM, Vol. 21(2), pp.120-126, February 1978.
[19] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," ACM Transactions on Computer Systems, Vol. 2(2), pp. 145-154, May 1984.
[20] F. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial," ACM Computing Surveys, 22(4), pp. 299-319, December 1990.
[21] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, and D.T. Seaton, "Fail-Controlled Computer Architectures for Distributed Systems" , University of Newcastle upon Tyne, Technical Report TR-333, July 1991.
[22] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, "Principal Features of the Voltan Family of Reliable Node Architectures for Distributed Systems," IEEE Transactions on Computers, (Special Issue on Fault-Tolerant Computing), 41(5), pp.542-549, May 1992.
[23] T.B. Smith, "Fault Tolerant Processor Concepts and Operation," Digest of Papers, FTCS-14, Kissimmee, USA, pp. 158-163, June 1984.
[24] N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan and S.K. Shrivastava, "The Design and Implementation of VOLTAN Fault-Tolerant Nodes for Distributed Systems," Transputer Communications, Vol. 1(2), pp. 93-109, November 1993.
[25] N. Theuretzbacher, "'VOTRICS': Voting Triple Modular Computing System," Digest of Papers, FTCS-16, Vienna, Austria, pp. 144-150, July 1986.
[26] A. Tully and S.K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs," Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, pp. 104-113, October 1990.
[27] S. Webber, and J. Beirne, "The Stratus Architecture," Digest of Papers, FTCS-21, Montréal, Canada, pp. 79-85, June 1991.
[28] J.H. Wensley et al, "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control," Proceedings of IEEE, Vol. 66(10), pp.1240-1255, October 1978.