Timeout Based Message Ordering Protocols for Software Implementation of TMR Systems

Paul D. Ezhilchelvan†, Francisco V. Brasileiro‡, and Neil A. Speirs†

†Department of Computing Science
University of Newcastle upon Tyne
Newcastle Upon Tyne, NE1 7RU, UK

‡Departamento de Sistemas e Computação
Universidade Federal da Paraíba - UFPb/Campus II
Av. Aprígio Veloso s/n, Campina Grande - Pb, 58100-000, Brazil

Abstract

Replicated processing with majority voting is a well known method for achieving reliability and availability - two key attributes of dependability. Triple Modular Redundant (TMR) processing is the most commonly used version of that method. Replicated processing requires that the replicas reach agreement on the order in which messages are to be processed. Almost all synchronous and deterministic ordering protocols published in the literature are time based in the sense that they require replicas' clocks to be kept synchronised within some known bound. We present a protocol for TMR systems that is based on timeouts and consequently does not require clocks to be kept in bounded synchronism. We analyse the protocol performance and show that this protocol is as fast as any time based ordering protocol when clocks cannot be synchronised using low-level processes with fast access to communication hardware. We also derive a faster protocol that has an improved performance in the absence of processor failures. We then build a TMR node and measure its performance to illustrate that the protocols developed here are useful for fast message ordering and are easy to implement.

Keywords and Phrases

Byzantine failures, fault tolerance, Triple Modular Redundancy (TMR), process replication, agreement, message ordering, physical and logical clocks.

1. Introduction

Designing and implementing a distributed system that continues to provide specified services in the presence of a bounded number of failures is a difficult task. One of the problems is that a faulty processor in a distributed system might fail in an arbitrary or Byzantine [Pease80] manner, the effects of which must be prevented from manifesting in the results delivered by the application programs. An approach to building a fault tolerant distributed system is to build processing sites or 'nodes' that mask the effects of processor failures, and then to build the distributed system using these failure-masking nodes. N modular redundant (NMR) nodes [Wensl78] are an example of nodes which can mask the failures of a bounded number of their internal components. The basic idea of N-modular redundancy is to use N, $N \geq 3$, processors to mask the failures of at most (N-

1)/2 processors. A Triple modular redundant (TMR) node with N=3 is the most practical version of an NMR node.

A TMR node consists of three directly connected processors which fail independent of each other and communicate with each other only by message passing. These processors execute every given task in parallel and the results produced by them are subject to a majority vote. The voted result will be regarded by the node environment as the output from the node. The TMR node will function correctly if at least two of its constituent processors function correctly.

Building a TMR node essentially involves synchronising the task execution on redundant processors and performing a majority vote on processors' outputs, and there are two widely-used approaches to achieve this. The hardware approach uses specialised hardware units such as: majority voter, bus interface and clock circuits. A common (reliable) clock source is used for driving the processors which execute in lock-step, with their outputs compared and sent out by a (reliable) voter. Alternatively, the node can be built using 'off-the-shelf' processors and relying purely on software implemented protocols for redundancy management and output voting. Each approach has its own advantages and limitations which we have discussed in [Brasil96]. Here, we consider solving an important problem in the software approach to building a TMR node.

Process replication within a TMR node assumes the well understood state machine model, for which precise requirements for supporting replicated processing are known [Sch90]. One of the requirements is that non-faulty processors of a TMR node process messages that are input to the node, in an identical order; i.e., non-faulty processors must *agree* on the relative *order* in which each input message is to be processed. In this paper, we develop order protocols to meet this requirement, in the following environment within a TMR node: (i) processors can sign and authenticate messages, and at most one processor can fail in Byzantine manner; (ii) non-faulty processors communicate within a known and bounded time; and, (iii) each non-faulty processor has a local read-only physical clock to measure time, and each clock's running rate with respect to the passage of real time[1] differs from unity by a small and known bound. Note that (ii) and (iii) make the environment synchronous; they are essential to guarantee that agreement on message ordering is reached in finite time in the presence of failures [Fisch85], and thus to estimate the bounds on the response times of the TMR node.

One of the difficulties in reaching agreement in the presence of failures is to provide processors with the ability to detect late or absent messages. This capability is usually achieved in a synchronous environment by synchronising non-faulty processors' clocks within some small and known bound. Let e denote this bound. Given that d is the known bound on message

1. Real time is assumed to be the mathematical Newtonian time that cannot be directly observed.

communication delays, a message m that is known to have been sent at time T according to the sending processor's clock, must be received no later than T + d + e according to the receiving processor's clock. In other words, if m is received after T + d + e, then m must be regarded as a late message by the receiving processor; if m is expected and not received until T + d + e, then m can be regarded as an absent message.

An alternative to synchronising processor clocks will be to have processors maintain Lamport's logical clocks [Lampo78]: a processor advances its logical clock when it sends a message and when it receives a message that appears to have been sent at the current or future time relative to its own logical clock. Consider a distributed system in which each processor relays every new message it receives, to every other processor. In such a system, logical clocks of any two non-faulty processors will exceed a given value within d time units of each other. Suppose that m is known to have been sent immediately after the sending processor's logical clock has exceeded the value T. This m must be received before an interval of 2d time elapses following the instance when the receiving processor's logical clock first exceeds T. In other words, the receiving processor should regard m as a late message if m arrives after the expiry of a 2d -timeout that was set soon after its logical clock has exceeded T. Thus, maintaining logical clocks and setting of timeouts can provide processors with a means to detect late and absent messages.

Lamport's subsequent work [Lampo84] presents a generic, Byzantine fault-tolerant solution which is based on synchronised clocks and is useful for solving not just the message ordering problem but also other fundamental problems of distributed computing such as distributed semaphore, transaction commit, etc. Following the terminology of [Lampo84], we will call the synchronised clock approach the time based approach, and the logical clock approach the timeout based one. [Lampo84] also demonstrates that the time based solutions work faster compared to the timeout based ones, if clocks can be synchronised using low-level processes with fast access and control over the communication hardware. This observation, in our opinion, has left the timeout based approach largely unexplored for developing agreement protocols tolerant of Byzantine faults.

We were motivated to consider timeout based message ordering for the construction of software implemented TMR node for the following three reasons. First, we are not interested in a generic distributed solution, but require an efficient message ordering protocol for a simple 3-processor distributed system. Secondly, when low-level processes cannot be used to synchronise clocks, the clock synchronisation bound can become large due to large uncertainties in message processing and queuing delays. If a TMR node is to be built out of 'off-the-shelf' processors running a given operating system, clock synchronisation has to be carried out by processes running above the operating system. This will inevitably result in poor clock synchronisation bounds, making the time based approach no more attractive than the timeout based approach. Finally, our approach of not relying on special purpose hardware means that the use of custom-built clock synchronisation unit, such as the clock synchronisation unit of MARS processors [Kopetz87], should be avoided. That is, clocks have to be synchronised using software. The programs used to achieve clock synchronisation tend to be complex and increase the overall complexity of the TMR software. To illustrate this, we will briefly describe the problems to be addressed in synchronising clocks.

Non-faulty processors' clocks do not normally have identical running rates and hence they can drift apart from each other. Despite this drift, the aim of clock synchronisation is to ensure that clock readings at any given instance of real time differ only within a known and fixed bound. This is to be achieved by having each processor periodically adjust its clock reading by an appropriate amount. That is, each processor should have a 'clock layer' that (i) executes periodically a clock synchronisation protocol to estimate the required amount of adjustment; (ii) provides the *abstraction* of a synchronised clock whose reading is the sum of the physical clock reading and the adjustment that is stored in a memory location, as the processor's physical clock is read-only and cannot be directly adjusted (see [Dolev84] for details); and (iii) 'phases in' the adjustment gradually to avoid sudden jumps or rollbacks in the readings of the synchronised clock. Thus, keeping processors' clocks in bounded synchronism involves estimating and phasing-in adjustments periodically. The software component that provides the above functionality naturally increases the overall complexity of the software, posing additional problems in testing and verification.

In this paper, we develop an order protocol that does not require the maintenance of synchronised-clock abstraction, but relies directly on physical clocks for knowing the current time and for scheduling operations at future times. A significant feature of our protocol is that the routing and timing information contained within a received message are used to minimise the message ordering delays. Note that the performance of an ordering protocol influences the node performance significantly. We show that our protocol works as fast as any time based protocol in an environment where processors' clocks can only be synchronised using high level software processes running on top of a standard operating system. The performance comparison of protocols is carried out both by theoretical analysis and by implementation. Our theoretical study makes some simplifying assumptions that favour only the performance of time based protocols. Despite these assumptions, analysis indicates that our protocol is equally fast. The measurements taken from our implemented system indicate that our protocol performs better. We then derive an 'early order' version of the protocol which has an improved performance in the absence of failures. This improvement is however achieved with an increase in message cost. This early order protocol would be particularly useful for message ordering within a (3-processor) *fail-stop* node [Sch84] where detection of a single internal failure causes the node to stop functioning and therefore message ordering is to be achieved mostly in a failure-free environment. The protocols presented here are an improved and more complete version of [Brasil95b]. The paper is organised as follows: Section 2 describes the structure of the TMR node and the assumptions made to develop the protocol. Section 3 develops and presents the protocol, proves its correctness; the protocol performance is analysed theoretically and compared with the performance of a time based protocol. In Section 4, the early order version of the protocol is derived. In Section 5 we present the implementation of a TMR node and compare the performance figures obtained for our protocols with those for the time based protocol. Section 6 concludes the paper.

2. System Description and Assumptions

We will suppose that the TMR node is made up of processors named P_i, P_j and P_k. These processors are uniquely ordered and the ordering is known to them. Each processor is connected directly to the other two processors of the node by *internal* links. Also, it is connected to the 'outside world' or the node environment from which input messages are received and output messages are sent to. Figure 1 shows a TMR node whose processors are connected to the environment via a bus.

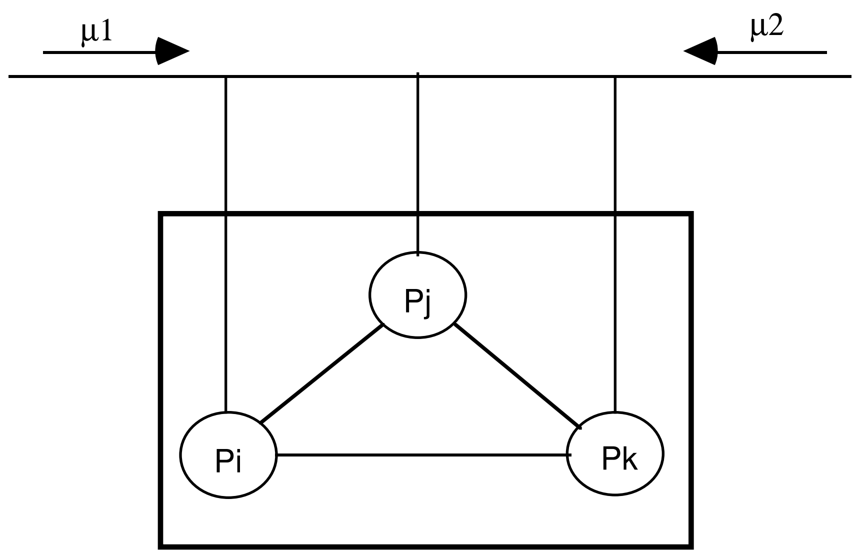

Figure 1. A TMR node.

Assumption 1: Within a TMR node, at least 2 out of 3 processors are non-faulty and never fail.

Assumption 2: The internal links that connect processors never fail and processors are reliably connected to the node environment. (Alternatively, a link failure can be thought of as the failure of one of the two processors connected by that link.)

Since one processor within a node can be faulty, any input message for the node must be transmitted to at least two processors. Note that there is no guarantee that non-faulty processors receive the input messages in the same order: referring to figure 1, P_i can receive μ_1 followed by μ_2 and P_k in the reverse order. So, whenever a processor receives an input message from the node environment, it forms an internal message m that contains the received input message in the field $m.\mu$. It decides the processing order for m (and thus for $m.\mu$) by sending m to all other processors in the node using a protocol that guarantees the following two conditions:

Validity: If a non-faulty P_i forms and sends m, all non-faulty processors (including P_i) will decide on an order for m, within a known and bounded real time interval Δ; and,

Unanimity: If a non-faulty P_i decides on an order for a given m, then every non-faulty P_j decides on the same order for m.

When each processor forms and sends a message m for a given input message, the input message gets ordered three times; all but the first agreed order are regarded as redundant and will be ignored.

Each processor can sign the messages it sends, and can authenticate the signed messages it receives. The following assumptions are made on the signature and authentication capabilities of non-faulty processors [Rivest78].

Assumption 3a: A non-faulty processor's signature for a given message is unique and cannot be generated by any other processor.

Assumption 3b: Any attempt to alter the contents of a non-faulty processor's signed message is detected by any other non-faulty processor.

In this paper we adopt the style of writing real time values in Greek or italicised upper case Roman letters, and clock time values in italicised lower case Roman letters; the term 'clock' will always refer to a processor's hardware clock.

Assumption 4: If a non-faulty P_i prepares and transmits m at real time T_i, every non-faulty destination P_j will receive m at real time T_j, $T_i \le T_j < (T_i+\delta)$, where δ, $\delta > 0$, is known.

Assumption 5: A non-faulty P_i's clock measures a clock time interval x in a real time interval $x(1+\rho_i)$, where $|\rho_i| \le \rho$ and ρ is a known positive constant.

3. The protocol

The development of the protocol mainly involves implementing (i) *diffusion* of messages to ensure that non-faulty processors receive an identical set of messages from a faulty processor, and (ii) *timeliness checks* to enable a non-faulty processor to assess the timeliness of a received message.

3.1. Message Counter and Diffusion

Each processor within the TMR node, say P_i, maintains a counter called the *message counter*, denoted as MC_i, which holds an integer value and is initialised to 1 when the node is first started. This counter is incremented in the same way as Lamport's logical clocks [Lampo78]. Before P_i sends the message m it formed, it performs the following actions on m: first, it time-stamps m by setting the message time-stamp field $m.TS = MC_i$ and increments MC_i by 1 - thus ensuring that any message it later forms gets a time-stamp larger than $m.TS$; it sets the message originator field $m.O$ to its identifier, i; and finally it generates a signature for m which is put in the signature field $m.S$. P_i then sends m to the other two processors in the node, and *accepts* m by entering a copy of m into a message list called the $accepted_i$.

Whenever P_i receives a message m from another processor, it checks whether m has been properly signed and is timely. (Procedures for checking the timeliness of a received message are described in the next sub-section.) If m is properly signed and timely, the following two actions are performed: (i) MC_i is set to the maximum of $\{MC_i, m.TS+1\}$ and the received m is *accepted* by entering a copy of m into $accepted_i$; and, (ii) if the accepted m has been signed just once, P_i generates a signature for m and appends the generated signature to the signature already in $m.S$; m is then sent to the third processor that has not signed it. Thus, any accepted message that is received with one signature is 'diffused' to the processor that appears not to have 'seen' the message. An accepted message is not diffused if it is received with signatures of two distinct processors.

For a given message m, we define *equiv*(m) as any m' such that $m'.\mu = m.\mu$, $m'.O = m.O$ and $m'.TS = m.TS$. Thus, an *equiv*(m) can be m itself or any other message that differs from m only in the signature field. As there can be at most one faulty processor in the TMR node, diffusion of single-signed messages ensures that if m enters $accepted_i$ of any non-faulty P_i, then at least one *equiv*(m) is received by the non-faulty P_j that did not form m, i.e., $m.O \neq$ j. This observation is proved as lemma 1 below.

Lemma 1: If a non-faulty P_i accepts m at real time T_i, then any other non-faulty P_k either sends or receives an authentic *equiv*(m) at real time T_k such that $|T_i - T_k| < \delta$.

Proof: Lemma 1 is true due to message diffusion and assumption 4. P_i accepting m could only be due to any one of the following events: (i) P_i forms and sends m; (ii) P_k forms and sends m; (iii) P_i receives a single-signed m from P_j; or (iv) P_k accepts a single-signed *equiv*(m) from P_j and then diffuses (double-signed) m to P_i. In all cases, the m accepted by non-faulty P_i must be authentic.

By assumption 4, $T_i \leq T_k < T_i + \delta$ and $T_k \leq T_i < T_k + \delta$ in cases (i) and (ii) respectively. In (iii), P_i diffuses an authentic, double-signed *equiv*(m) to P_k which is received at some time T_k, $T_i \leq T_k < T_i + \delta$. In the case of (iv), P_k must receive an authentic single-signed *equiv*(m) from P_j at time T_k, $T_i - \delta < T_k \leq T_i$, for its diffused message to be received and accepted by P_i at T_i. So, in all cases $|T_i - T_k| < \delta$ and the *equiv*(m) sent or received by P_k is authentic; hence the lemma. q

We will assume that each processor has the following two primitives to implement message authentication and diffusion: the *receive*(m) primitive blocks until it can return an authentic message m that is received via an internal link and has the signatures of one or two distinct processors other than the host processor; the *send*(m) primitive does nothing if m is double-signed and performs the following actions if m contains less than two signatures: it generates the sending processor's signature for m, appends the generated signature to any signature that may already be in $m.S$ and transmits the resulting message to processors in the node that have not signed m.

From the descriptions of these primitives, it is obvious that a message sent or received by a non-faulty processor will have at least one or at most two authentic signatures; every sent message, but no received message, carries the host processor's signature. For a given message m, *path*(m) is defined as the sequence of processors in the order they have signed m. Thus, if m is a double-signed message that is formed by P_j and diffused by P_k, then *path*(m) = P_j:P_k. The first processor in *path*(m) is called the *originator* of m and the last processor the *immediate_sender* of m. Note that the *originator* and the *immediate_sender* of m are one and the same if m is single-signed and that they are distinct for a double-signed m.

3.2. Timeliness Checks

These checks enable a processor to determine the timeliness of a received m. Before presenting them, we will define a clock time interval d such that by measuring d in its local clock a non-faulty processor is guaranteed to measure a real time interval of at least δ duration, i.e., $d \geq \delta/(1-\rho)$; d is known to all non-faulty processors of the node. We will assume, for simplicity, that a processor takes zero time to execute any instruction of the protocol and the *send*(m) and *receive*(m) primitives. (Realising this assumption in practice will require an increase in the value of d, which is possible as the protocol does not impose any upper bound on the value of d.).

Let us fix the processor P_i in the TMR node to be non-faulty. Supposing that P_i receives a message m at its local clock time t_i, we will describe the checks by which P_i should determine the timeliness of m. When P_i receives m at its local clock time t_i, there can arise one of three possible situations depending on the value of MC_i at that time: If $MC_i < m.TS$ or $MC_i = m.TS$ then m is a 'future' or a 'present' message respectively and must be considered by P_i as timely; if, on the other hand, MC_i is already larger than $m.TS$ when m is being received, then m is a 'past' message, and its timeliness will depend on how much time has elapsed since MC_i first became larger than $m.TS$. So, we have to design timeliness checks only for messages received with past timestamps. Let m', $m' \neq m$, be the message whose acceptance by P_i caused MC_i to become larger than $m.TS$ for the first time. That is, just before m' is to be accepted, $MC_i \leq m.TS$. Note that P_i could have either received m' or formed and sent it, and also that $m'.TS \geq m.TS$ is true because acceptance of m' has caused MC_i to exceed $m.TS$. Suppose that P_i accepts m' at local clock time t'_i, $t'_i < t_i$. For m to be considered timely, $(t_i - t'_i)$ must be less than a certain bound called the *timeliness bound* whose length depends on *path*(m') and *path*(m). Table 1 below summarises the timeliness bounds used by P_i for various combinations of *path*(m') and *path*(m).

$path(m)$ / $path(m')$	P_k (1)	P_j (2)	$P_j{:}P_k$ (3)	$P_k{:}P_j$ (4)
P_i (A)	$2d$	$2d$	$4d$	$4d$
P_k (B)	d	$2d$	$3d$	$3d$
P_j (C)	$2d$	d	$3d$	$3d$
$P_j{:}P_k$ (D)	d	d	$2d$	$3d$
$P_k{:}P_j$ (E)	d	d	$3d$	$2d$

Table 1: P_i's timeliness bounds for m given that MC_i first exceeded $m.TS$ due to accepting m'.

3.3. Establishing Timeliness Bounds

In this subsection, we will describe how the bounds of Table 1 have been arrived at. To keep the description simple, we will make the following three assumptions: (i) The clocks of all non-faulty processors have identical running rates. (This assumption is later removed in this subsection.) (ii) Unless stated explicitly, time is measured according to the clock of the non-faulty processor P_i. So, when we say an event happened at (time) t, it means that the event happened at time t according to P_i's clock. (iii) Finally, where the context is obvious, the subscript i is dropped from t_i and t'_i which are respectively the reception time of m and the acceptance time of m' by P_i.

From the descriptions of m and m', we have: $t' < t$, $m'.TS \geq m.TS$ and $MC_i \leq m.TS$ just before t'. Note that each timeliness bound in Table 1 must be estimated to be as *small* as possible so that the overall message ordering delay is small, and also must satisfy the following *safety* property:

safety: P_i must find m timely if the *immediate_sender* of m is non-faulty.

This safety property ensures that non-faulty processors do not reject each other's messages as untimely. To estimate the timeliness bounds that are small and safe, we first extend lemma 1 to lemma 2.

Lemma 2: That P_i accepts m' at t' implies that any other non-faulty P_k will have $MC_k > m'.TS$ at $t' + d$.

Proof: By lemma 1 and by the definition of d, P_k sends or receives an (authentic) *equiv*(m') before $t'+d$. If $MC_k \leq m'.TS$ when P_k receives the *equiv*(m'), it must unconditionally accept the received message which will set $MC_k > m'.TS$. Hence the lemma. q

The algorithm for sending a locally formed message ensures that when a processor forms and sends m, $m.TS$ is set to the current value of MC. Therefore the following can be stated: if P_i receives a single-signed m, $m.TS \leq m'.TS$, after $t' + 2d$, then, by lemma 2 and by the definition of d, the sender of m cannot be non-faulty. In other words, P_i must consider a received m timely only if m is received before $t' + 2d$; i.e., for a single-signed m to be regarded as timely, $(t - t') < 2d$

must be true irrespective of *path*(m) and *path*(m'). Generalising this to any non-faulty processor, we state the first of the rules which will form the basis for checking message timeliness.

Rule 1: A processor should consider a single-signed message *msg* as timely, only if *msg* arrives in less than $2d$ time after its *MC* first exceeds *msg.TS*, where time is measured according to the processor's clock. m

Let m be a double-signed message. As per lemma 2, P_i knows that the MC of the *immediate_sender* of m exceeds $m.TS$, $m.TS \le m'.TS$, before $t' + d$. By rule 1, the *immediate_sender* of m must not accept the single-signed *equiv*(m) from the *originator* of m at or after $t' + d + 2d$. So, if the *immediate_sender* of m is non-faulty, P_i must receive m before $t' + d + 2d + d$. So, P_i should regard a double-signed m as timely, only if $(t - t') < 4d$. In other words, if $t > t' + 4d$, then the *immediate_sender* of m must be faulty and m must be regarded as late. This fact is generalised as rule 2.

Rule 2: A processor should consider a double-signed message *msg* as timely, only if *msg* arrives in less than $4d$ time after its *MC* first exceeds *msg.TS*, where time is measured according to the processor's clock. m

Note that rules 1 and 2 are trivially satisfied for any *msg* formed and sent by P_i itself: a locally formed *msg* 'arrives' in almost zero time after MC_i first exceeds *msg.TS*. The timeliness bounds suggested by these rules are reflected in the Ath row of Table 1 (which represents all cases where the *originator* of m' is P_i itself), and the entries in the Bth row of the second column (entry B2) and in the Cth row of the first column (entry C1). For all other entries, the bounds suggested by the two rules are not incorrect but are an over-estimation and can be reduced by at least d. This reduction is achieved by exploiting the knowledge which P_i can derive when it receives m: any non-faulty processor in the *path*(m) has its *MC* larger than *m.TS* at the time P_i receives m; further, if m is double-signed then *all* non-faulty processors in the node have their *MC* larger than *m.TS* when P_i accepts m. We will discuss this reduction by first refining lemma 2 for cases where P_i is not the *originator* of m'.

Lemma 3: That P_i receives and accepts m' at t' implies that

(i) if the *originator* of m' is non-faulty, it will have the value of its *MC* exceed $m'.TS$ at or before t'; and,

(ii) if the *immediate_sender* of m' is non-faulty, it will have the value of its *MC* exceed $m'.TS$ at or before t'.

Proof: (i) implies (ii) if m' is a single-signed message. The *MC* of the *originator* of m' becomes larger than $m'.TS$ soon after m' is timestamped; if m' is a double-signed message, *MC* of the *immediate_sender* of m becomes larger than $m'.TS$ before m' is diffused to P_i. Recall that t' is the time when P_i receives m', and that message processing and transmission can take a minimum of zero time. So, *MC*s of the *originator* and the *immediate_sender* of *m*' should exceed *m*'.*TS* at or before *t*'. Hence the lemma. q

Suppose that the message *m* received by P_i at *t* is single-signed, and that the *originator* of *m* is the *originator* or the *immediate_sender* of *m*'. By lemma 3, P_i which received *m*' at *t*', *t*' < *t*, knows that the *MC* of the *originator* of *m* has become larger than *m*'.*TS,* no later than *t*'. Since *m*.*TS* ≤ *m*'.*TS*, the *originator* of *m* must have formed and sent *m* before *t*'. So, by the definition of *d* and assumption 4, (*t* - *t*') < *d* for *m* to be timely. From this observation the third rule is derived:

Rule 3: Consider a processor that receives a single-signed *msg* after its *MC* has exceeded *msg.TS* due to having received and accepted a message *msg*'. If the *originator* of *msg* is the *originator* or the *immediate_sender* of *msg*', then the processor should consider *msg* timely only if *msg* is received in less than *d* time after accepting *msg*', where time is measured according to the processor's clock. m

Since the total number of processors in the TMR node is three, if *msg*' in rule 3 is double-signed, then the *originator* of *msg* will have to be either the *originator* or the *immediate_sender* of *msg*'. So, a variation of rule 3 can be stated when *msg*' is double-signed:

Rule 4: Consider a processor that receives a single-signed *msg* after its *MC* has exceeded *msg.TS* due to having accepted a double-signed *msg*'. It should consider *msg* timely only if *msg* is received in less than *d* time after accepting *msg*', with time being measured according to the processor's clock. m

The *timeliness bound* of *d* indicated in the entries B1, D1, E1, C2, D2, and E2 of Table 1 are due to rules 3 and 4. This completes the case where *m* is a single-signed message, leaving us to consider reducing the *timeliness bounds* for double-signed *m* when *path*(*m*') ≠ P_i. Recalling that the *originator* and the *immediate_sender* of the double-signed *m* are different, we will consider three disjoint categories which together cover all possible combinations of the paths of a single- or double-signed *m*' and a double-signed *m*. These categories are: (i) *path*(*m*) = *path*(*m*'), (ii) the *immediate_sender* of *m* = the *originator* of *m*', and (iii) the *originator* of *m* = the *originator* of *m*' and *path*(*m*) ≠ *path*(*m*').

Category 3.1: when *path*(*m*) = *path*(*m*'), the *originator* of *m*' = the *originator* of *m* = P_j (say), and the *immediate_sender* of *m*' = the *immediate_sender* of *m* = P_k.

Let us fix P_k, the *immediate_sender* of *m*, to be non-faulty to ensure that the timeliness bound estimated is safe. If P_k accepts the single-signed *m* first and then the single-signed *m*', then P_i should receive *m* in less than *d* time after it receives *m*'; that is, (*t* - *t*') < *d*. (Note that no *fifo* communication between processors is assumed and so messages transmitted within δ real time can overtake each other during transmission.) Suppose that P_k accepts the single-signed *m*' before the single-signed *m*. By lemma 3, MC_k exceeds *m*'.*TS* no later than *t*'. By rule 3, P_k will not accept the single-signed m from P_j at or after $t'+d$. Therefore, when P_k is non-faulty, P_i should receive m before $t'+2d$; i.e., m is timely only if $(t - t') < 2d$. Choosing the largest of the bounds estimated for the two possibilities (of P_k accepting m before or after m'), $2d$ becomes the timeliness bound for this category which is indicated by the entries in D3 and E4 of Table 1. w

Category 3.2: the *immediate_sender* of m = the *originator* of $m' = P_k$ (say).

Note that the message m' that P_i accepts can be single- or double-signed. As before, let us fix P_k, the *immediate_sender* of m, to be non-faulty. If P_k first accepts the single-signed *equiv*(m) from P_j and then forms and sends *equiv*(m'), then P_i should receive m in less than d time after it receives m'; that is, $(t - t') < d$. Suppose that P_k accepts the single-signed *equiv*(m) from P_j after it formed and sent *equiv*(m'). By lemma 3, MC_k exceeds $m'.TS$ no later than t' which is the time when P_i accepts m'. By rule 1, P_k will not accept the single-signed m from the *originator* of m at or after $t'+2d$. Therefore, P_i should receive the diffused m from P_k before $t'+2d+d$; i.e., m is timely only if $(t - t') < 3d$. Choosing the largest of the bounds estimated for the two possibilities (of P_k accepting *equiv*(m) before or after sending *equiv*(m')), $3d$ becomes the timeliness bound that appears in entries B3 and E3 (where the *originator* of m' is P_k), and in the entries C4 and D4 (where the *originator* of m' is P_j). w

Category 3.3: the *originator* of m = the *originator* of $m' = P_k$ (say) and *path*(m) ≠ *path*(m').

Let us suppose that P_j, the *immediate_sender* of m, is non-faulty. (Note: P_k here need not be non-faulty.) Since *path*(m) ≠ *path*(m') and m is a double-signed message, m' must be a single-signed message. So, P_i diffuses m' which it accepts, to P_j and the diffused message must reach P_j before $t'+d$. P_j can receive the diffused double-signed m' either before or after it receives the single-signed m from P_k. Say, P_j receives the single-signed m from P_k before it receives m' from P_i, i.e., no later than $t'+d$. In that case, P_i must receive the diffused m from P_j no later than $t'+d+d$; that is, $(t - t') < 2d$.

Let us consider the case where P_j receives the single-signed m from P_k after it has received the double-signed m' from P_i. Since P_j finds the single-signed m timely and $m.TS \le m'.TS$, by the Lemma 4 stated below, it must also find the double-signed m' timely. By rule 4, P_j must receive the single-signed m within d time after it receives the double-signed m'; otherwise it would have rejected the single-signed m as untimely and not have diffused it to P_i. Since P_j can receive the double-signed m' from P_i just before $t'+d$, it could receive the single-signed m before $t'+d+d$. So, P_i could receive the diffused m from P_j before $t'+d+d+d$. Thus, $(t - t') < 3d$ is possible.

Choosing the largest of the bounds estimated for the two possibilities (of P_j receiving the single-signed m from P_k before or after receiving the double-signed m' from P_i), $3d$ becomes the timeliness bound for this category which is indicated by the entries in B4 and C3 of Table 1. w

Lemma 4: Let t_1 and t_2 be the local clock times when a non-faulty P_i receives a single-signed m_1 and a double-signed m_2, respectively; also, let $m_1.TS \leq m_2.TS$ and $t_1 > t_2$. If P_i finds m_1 timely then it must also find m_2 timely.

Proof: We will measure time according to P_i's local clock and prove the lemma by contradiction. Suppose that P_i finds m_2 late. Let m_2 be late by y, $y > 0$, time units, i.e., if P_i had received m_2 at any time before $(t_2\text{-}y)$ then it would have found m_2 timely. That m_2 received at t_2 was found late implies that there exists a message m', $m'.TS \geq m_2.TS$, which was accepted by P_i at $(t_2 - y - tb_2)$, where tb_2 is the timeliness bound indicated by the entry of Table 1 whose row corresponds to the *path*(m') and column to the *path*(m_2). Let tb_1 is the timeliness bound indicated by the entry of Table 1 whose row corresponds to the *path*(m') and column to the *path*(m_1).
Since $m_1.TS \leq m_2.TS \leq m'.TS$, m_1 must be received by P_i before $(t_2 - y - tb_2 + tb_1)$ for it to be considered timely. For any given *path*(m'), i.e. in any given row of Table 1, the timeliness bound for a single-signed message is smaller than that for a double-signed-message. That is, $tb_1 < tb_2$. This means, $(t_2 - y - tb_2 + tb_1) < t_2$; by given, $t_2 < t_1$. So, $t_1 > (t_2 - y - tb_2 + tb_1)$. This means that m_1 is received by P_i after $(t_2 - y - tb_2 + tb_1)$ and cannot be found timely by P_i. This is a contradiction. q

Next we remove the assumption that non-faulty processors' clocks have identical running rates. When the readings of non-faulty clocks can drift apart with the passing of real time, the differences accumulated over a period of time must be compensated so that the safety property of the timeliness bounds is not compromised. The way we do this compensation is by increasing the value of d by the maximum difference that could accumulate during the 'life-time' of a non-faulty processor's message within the ordering protocol, i.e., during the maximum real-time within which a non-faulty processor's message gets ordered by every other non-faulty processor in the node. Section 3.5 estimates this life-time to be not larger than $4d(1+\rho)$. So, $d \geq \delta/(1-\rho) + (2\rho)4d$, where the first component ensures that at least δ real time passes during d clock time and the other component is the maximum clock difference that can develop over a period of $4d(1+\rho)$. Ignoring the second and higher order powers of ρ, we get: $d \geq \delta/(1-9\rho)$. However, a closer analysis indicates that the case that determines the maximum ordering delay is not the one that determines the smallest lower bound on d and the least stringent requirement on d turns out to be: $d \geq \delta/(1-5\rho)$. The appendix presents proof to show that $\delta/(1-5\rho)$ is the smallest and safe lower bound on d for the set of timeliness bounds in Table 1. Below, we present the scenario which gives rise to that bound on d:

Suppose that non-faulty P_k's clock is running fast with $\rho_k = -\rho$, and non-faulty P_i's clock is running slow with $\rho_i = \rho$. Let P_i form and send m' at local clock time t_i', and let m' be transmitted in zero time; let P_k receive m' at its clock time t_k'. Suppose that a faulty P_j forms and sends m only to P_i, and that P_i receives and accepts single-signed m from P_j just before its clock time $t_i'+2d$ and then diffuses the double-signed m to P_k. P_k's clock, despite running fast, should not be reading past $t_k'+3d$ (see entry B3 of Table 1 with suffix i replaced by k) when the diffused double-signed m arrives from P_i, even if the transmission from P_i to P_k is to take just less than δ - the maximum permitted time (see assumption 4). So, d must be large enough to ensure that $3d(1+\rho_k) \geq 2d(1+\rho_i)+\delta$; that is $d \geq \delta/(1-5\rho)$.

3.4. Protocol Description

To implement the timeliness checks discussed above, each non-faulty P_i maintains path counters, denoted as $PC_i[p]$, for every path p through which P_i can receive a message. Thus, P_i maintains four path counters, one for each p in $\{P_k, P_j, P_j{:}P_k, P_k{:}P_j\}$. These counters have integer values which are initialised to zero when the node is formed. A path counter $PC_i[p]$ is set to ts, to indicate that any m, $m.TS \leq ts$ and $path(m) = p$, to be received thereafter, must not be considered timely. A process, named *update*, is used to maintain these counters. Its algorithm is given below:

```
update(path: p, timestamp: ts)
{
  PCi[p] = max{PCi[p], ts}
}
```

Whenever P_i forms and sends m' or receives a timely m', executions of the *update* process are scheduled to set $PC_i[p] = max\{PC_i[p], m'.TS\}$, for every $p \in \{P_k, P_j, P_j{:}P_k, P_k{:}P_j\}$ after the elapse of time intervals indicated by the row of Table 1 that corresponds to $path(m')$. For example, when P_i forms and sends m' at its clock time t_i', executions of the *update* process are scheduled to set $PC_i[p] = max\{PC_i[p], m'.TS\}$ at clock times $t_i'+2d$, $t_i'+2d$, $t_i'+4d$, and $t_i'+4d$ for p equal to P_k, P_j, $P_j{:}P_k$ and $P_k{:}P_j$, respectively (row A); similarly, if P_i receives and accepts a timely m' at t_i' with path(m') = $P_j{:}P_k$, then executions of the *update* process are scheduled to set $PC_i[p] = max\{PC_i[p], m'.TS\}$ at clock times $t_i'+d$, $t_i'+d$, $t_i'+2d$, and $t_i'+3d$ for p equal to P_k, P_j, $P_j{:}P_k$ and $P_k{:}P_j$, respectively (row D). The executions of the *update* process update the path counters at appropriate timing instances and the timeliness of a received message can be easily checked by referring to the path counter corresponding to the path of the received message.

We define $PC_{i,mn}$ = minimum of $\{PC_i[p] \mid \forall p, p \in \{P_k, P_j, P_j{:}P_k, P_k{:}P_j\}\}$. Any m that is received after $PC_{i,mn}$ has become larger than or equal to $m.TS$ is untimely irrespective of $path(m)$ and cannot enter $accepted_i$, i.e., $accepted_i$ becomes closed for such m arriving after $PC_{i,mn} \geq m.TS$. So, after $PC_{i,mn} \geq ts$, every m, $m.TS \leq ts$, that is already in $accepted_i$ becomes 'stable' and can be safely ordered. Stable messages of identical timestamp are grouped together and transferred into another list called $stable_i$ for ordering. Duplicates in $stable_i$ are removed by replacing more than one $equiv(m)$ with just one of those *equiv* messages. Two messages m and m' are said to be *spurious* if $m.O = m'.O$, $m.TS = m'.TS$ and $m.\mu \neq m'.\mu$. Spurious messages must have been formed and sent by a faulty processor that failed by giving the same timestamp to distinct messages m and m'. They are identified and removed from $stable_i$ before message ordering. An ordering relation « for distinct m and m' is defined: m « m', if (($m.TS < m'.TS$) **or** ($m.TS = m'.TS$ **and** $m.O < m'.O$)). Distinct and non-spurious m and m' in $stable_i$ are ordered according to the relation «. The protocol for P_i is presented below:

```
do /* Broadcast process handling messages received from outside the node */
  get M from outside the node;
  m.μ = M; m.TS = MCi; m.O = i;
  MCi = MCi +1;
  t = clocki;
  schedule update(PCi[Pk], m.TS) at t+2d;          /* implements A1 */
  schedule update(PCi[Pj], m.TS) at t+2d;          /* implements  A2 */
  schedule update(PCi[Pj:Pk], m.TS) at t+4d;             /* A3 */
  schedule update(PCi[Pk:Pj], m.TS) at t+4d;             /* A4 */
  send(m);
  acceptedi = acceptedi ∪ {m};
end do    /* Broadcast process */
||
do /* Diffuse process handling messages received via internal links */
  receive(m);
  if (m.TS ≤ PCi[path(m)] then discard(m)
  else
    MCi = max{MCi, m.TS+1};
    t = clocki;
    case path(m) == Pk
      schedule update(PCi[Pk], m.TS) at t+d;                 /* B1 */
      schedule update(PCi[Pj], m.TS) at t+2d;                /* B2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+3d;             /* B3  */
      schedule update(PCi[Pk:Pj], m.TS) at t+3d;             /* B4 */
    case path(m) == Pj
      schedule update(PCi[Pk], m.TS) at t+2d;                /* C1 */
      schedule update(PCi[Pj], m.TS) at t+d;                 /* C2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+3d;             /* C3 */
      schedule update(PCi[Pk:Pj], m.TS) at t+3d;             /* C4 */
    case path(m) == Pj:Pk
      schedule update(PCi[Pk], m.TS) at t+d;                 /* D1 */
      schedule update(PCi[Pj], m.TS) at t+d;                 /* D2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+2d;             /* D3 */
      schedule update(PCi[Pk:Pj], m.TS) at t+3d;             /* D4 */
    case path(m) == Pk:Pj
      schedule update(PCi[Pk], m.TS) at t+d;                 /* E1 */
      schedule update(PCi[Pj], m.TS) at t+d;                 /* E2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+3d;             /* E3 */
      schedule update(PCi[Pk:Pj], m.TS) at t+2d;             /* E4 */
    end case
    send(m);
    acceptedi = acceptedi ∪ {m}
  end if
end do    /* Diffuse process */
||
do  /* Order process identifying and ordering stable messages  */
  PCi,mn = min{PCi[p] | ∀ p: p ∈ {Pk, Pj, Pj:Pk, Pk:Pj}}
  while SCi < PCi,mn do
    SCi = SCi + 1;
```

```
        stable_i = {m | m ∈ accepted_i and m.TS = SC_i};
        accepted_i = accepted_i-stable_i;
        if stable_i ≠ {} then
          distinct = { };
          for every m in stable_i:
            if equiv(m) ∉ distinct then insert m in distinct end if
          end for
          stable_i = distinct;
          spurious = {m, m' | m∈ stable_i ∧ m'∈ stable_i ∧ m'.O=m.O ∧ m'.µ≠m.µ};
          stable_i = stable_i-spurious;
            if stable_i ≠ { } then order and deliver each m in stable_i according to « relation end if;
        end if
      end while
    end do    /* Order process */
```

Figure 2. The Order Protocol.

The protocol is implemented by three concurrent processes, namely *Broadcast*, *Diffuse* and *Order*, which respectively deal with: input messages received from outside the node; internal messages received from an internal link; and message ordering. The *Order* process transfers stable messages into $stable_i$ in the increasing order of their timestamps; for this ordered transfer it maintains a *stability counter* SC_i which is initialised to the initial value of $PC_{i,mn}$ and is incremented by 1 so long as SC_i is smaller than $PC_{i,mn}$. Following every increment of SC_i, it removes duplicate *equiv*(m) messages and any spurious messages from $stable_i$, and then orders the remaining messages according to the « relation: if m « m', m is ordered and delivered before m'.

An efficient implementation of the *Diffuse* process will avoid the need to remove duplicate *equiv*(m) messages by entering a received and timely m into $accepted_i$, only if an *equiv*(m) is not already in the list. (Note that this optimisation does not change the timeliness bounds, but merely redefines the notion of 'accepted' messages as messages which are entered into $accepted_i$ or cannot be entered only because an *equiv* message has already been entered.)

3.5. Protocol Correctness

Theorem 1: The protocol guarantees unanimity and validity conditions with $\Delta = 4d(1+\rho)$, provided $d \geq \delta/(1-5\rho)$.

Proof: Let P_i and P_k be any two distinct non-faulty processors in a TMR node. Let P_i form and send a message m at real-time T_i. P_k receives m before $T_i + \delta$ (by assumption 4). Lemmas S1 and S2 in the appendix prove that the timeliness bounds of Table 1 with $d \geq \delta/(1-5\rho)$ are safe. So, P_k will find the received m timely and accept it. Note that when a non-faulty processor accepts a message *msg*, none of its path counters stays below *msg.TS* after some finite time. So, the message m accepted by P_i and P_k, will be taken up for ordering and will be ordered because a faulty processor cannot form and send another message that could make P_i and P_k consider m as spurious (due to assumption 3). This shows that the validity condition is met for some finite Δ. The set of *schedule* instructions in the *Broadcast* process indicates that $PC_{i,mn}$

reaches $m.TS$ within $4d$ clock time after T_i. So, P_i orders m no later than $T_i + 4d(1+\rho_i)$. The set of *schedule* instructions in the *Diffuse* process indicates that $PC_{k,mn}$ reaches $m.TS$ within $3d$ clock time after P_k receives m which is before $T_i + \delta$. So, P_k orders m no later than $T_i + \delta + 3d(1+\rho_k)$. Thus the message m sent by P_i gets ordered by P_i and P_k, no later than T_i + maximum of $\{4d(1+\rho_i), \delta+3d(1+\rho_k)\}$. By assumption 5, $|\rho_i| \leq \rho$ and $|\rho_k| \leq \rho$. So, Δ = maximum of $\{4d(1+\rho), \delta+3d(1+\rho)\} = 4d(1+\rho)$.

Say, P_i receives and accepts m'. If m' is double-signed, then an $equiv(m')$ is already accepted by P_k. If m' is single-signed and not signed by P_k, then P_i will diffuse a double-signed m' to P_k which will accept the diffused message due to the safety property of the timeliness bounds. Thus, what is received and accepted by P_i is accepted by P_k; also, since the validity condition is met, what is sent by P_i is accepted by P_k. This means that non-faulty processors accept an identical set of messages for every given timestamp ts, $ts \geq 1$. Recall that when a non-faulty processor's stability counter becomes equal to ts, no msg with $msg.TS \leq ts$, will ever be accepted by that processor and every accepted msg' with $msg'.TS = ts$, enters the list *stable*. This implies that when SC_i and SC_k become $m'.TS$, P_i and P_k (i) enter an identical set of messages with timestamp $m'.TS$, and (ii) would have entered an identical set of messages with timestamp less than $m'.TS$. Because of (i), P_i finds m', $m' \in stable_i$, spurious if and only if P_k finds $equiv(m')$, $equiv(m') \in stable_k$, spurious; that is, P_i decides to order m' in $stable_i$ if and only if P_k decides to order $equiv(m')$ in $stable_k$. Say, P_i and P_k decide to order m' and the ordering will be according to the relation «. Since « provides total ordering on a given set of messages and because of (i) and (ii), for a given m' and any m, $m \neq m'$: P_i orders and delivers m before m' if and only if P_k orders and delivers $equiv(m)$ before $equiv(m')$. Thus the protocol satisfies the unanimity condition. q

3.6. Theoretical Analysis of Relative Performance

The Δ of theorem 1 is the *ordering delay* of the protocol which indicates the maximum real time interval which a non-faulty processor's message can take before it gets ordered by every non-faulty processor in the node. Note that the ordering delay cannot be, and is not, defined for m of a faulty sender, as m may never be ordered by non-faulty processors. Let us denote the ordering delay of a time based protocol as Δ'. [Dolev82] reports that reaching agreement in the presence of at most π faulty processors requires at most π+1 rounds of message exchange. Thus, with a time based protocol, if a non-faulty P_i forms and sends a message m at its synchronised clock time t_i, then every non-faulty processor orders m at its respective synchronised clock time $t_i+2(d+e)$, where $d \geq \delta/(1-\rho')$, ρ' is the maximum rate at which a non-faulty processor's synchronised clock can deviate from real-time and e is the precision of the clock synchronisation protocol (see [Cri85] for example). While the sender of m orders the sent message exactly at $t_i+2(d+e)$ according to its synchronised clock, any other non-faulty P_k orders m at some time between $t_i+2(d+e)-e$ and

$t_i+2(d+e)+e$ according to P_i's synchronised clock. So, $\Delta' = 2(d+e)(1+\rho')$ when P_i's clock is behind the clocks of all other non-faulty processors (the best case) and $\Delta' = 2(d+e)(1+\rho')+e$ when the clock of at least one non-faulty processor is e units of time behind P_i's clock (the worst case). As the readings of a synchronised clock are periodically adjusted, they will not be continuous and $\rho' > \rho$, unless special arrangements are made. To make the comparison simpler, we assume the use of amortization techniques [Schmuck90] to ensure continuity in the readings of synchronised clocks without increasing ρ', and also a mechanism that makes $\rho' = \rho$ (see [Srikanth85]).

Synchronisation precision	Δ'-Δ in the best case for Δ'	Δ'-Δ in the worst case for Δ'
$e = \delta/(1+\rho)$	$-12\delta\rho/(1-6\rho)$	$(1-26\rho)\delta/(1-5\rho)$
$e = (1+5\rho)\delta/(1+\rho)$	$-2\delta\rho/(1-6\rho)$	$(1-11\rho)\delta/(1-5\rho)$

Table 2: Δ'- Δ for different clock synchronisation methods

The exact value of e depends on the algorithm used to compute the adjustment, the frequency of clock adjustment, and how fast the processes of clock synchronisation protocol can access the internal links. Adjustments can be computed either by referring to a reliable clock that is external to the node, or by using authenticated clock synchronisation protocols, such as [Halp84, Lampo85, Srikanth85], that are suitable for TMR nodes. We ignore for simplicity (and in favour of synchronised-time based protocols) the effect of the frequency of clock adjustment on e; i.e. we ignore a component of e that accounts for clock drift until next adjustment and take e to be the maximum clock difference immediately after adjustments were made. Assuming that clock synchronisation protocol is implemented using high level processes, the value of e considered for comparison is $\delta/(1+\rho)$, $\delta/(1+\rho)$ and $(1+5\rho)\delta/(1+\rho)$, when an external clock, [Halp84] and [Srikanth85] are used respectively to synchronise clocks. (We omit the authenticated protocol of [Lampo85] as it is admittedly less efficient than [Halp84].)

Table 2 shows the (theoretical) difference Δ'-Δ in the best and worst cases for the time based protocols. Note that this comparison was made with conditions that only favour the synchronised-time based protocols. Despite this, our protocol is slow by $O(\delta\rho)$ (in the order of tens of picoseconds) only in the best case for Δ'; in addition the comparison does not consider the overhead for clock synchronisation and amortization. Further comparison of protocol performance is made in Section 5 which presents the ordering delays observed through an implementation of these protocols.

4. Early-Order Protocol

When processors of a TMR node are fully and reliably connected (as we have assumed here), it is easy to implement an inter-processor communication service which guarantees that messages sent by a non-faulty processor over an internal link are received in the sent order by the non-faulty processor at the other end of the link. Assuming that the processor communication within the TMR node has this *first-in-first-out* (*fifo*) property, we present a variation of the protocol discussed in the previous section which is guaranteed to take less than Δ time for ordering messages in the absence of failures. Due to this early ordering capability in failure-free situations, we call this protocol an *early-order* protocol.

The early ordering capability comes with a cost: processors must be lively and be sending messages frequently. We meet this requirement by forcing a processor to form and send a null message whenever it accepts a received message. A null message has empty $m.\mu$, i.e. $m.\mu = \phi$; where distinction is required, messages with $m.\mu \neq \phi$ will be called non-null messages. A null message is subject to the same timeliness and authentication checks as a non-null message, and is diffused when it is accepted with a single signature; the only difference in the treatment of a null message is that another null message is not formed and sent following its acceptance. By sending a null message in response to having accepted a non-null m', a processor indicates to other processors that its message counter has become larger than $m'.TS$; this in turn causes PC_{mn} of every processor to increase faster, thus permitting early ordering of messages.

4.1. New Timeliness Bounds

When fifo communication is not assumed, if two messages are sent by a non-faulty processor over a given link within δ time of each other, then it is possible that the second message overtakes the first message during transmission and is received first by the processor at the other end of the link. Fifo communication forbids this message overtaking and ensures that messages are received across the link in the transmitted order. Given that MC never decreases in value, the messages transmitted by a non-faulty processor over a given link will have their timestamps in the increasing order except when a double-signed message with timestamp smaller than MC is diffused. This fact allows the timeliness bounds of table 1 to be reduced to zero in some cases which are described here. In all other cases, the bounds of table 1 remain the same and table 3 below presents the safe and small timeliness bounds that can be used when the inter-processor communication satisfies the fifo property.

path(m) / *path*(m')	P_k (1)	P_j (2)	P_j:P_k (3)	P_k:P_j (4)
P_i (A)	$2d$	$2d$	$4d$	$4d$
P_k (B)	0	$2d$	$3d$	$3d$
P_j (C)	$2d$	0	$3d$	$3d$
P_j:P_k (D)	0	d	0	$3d$
P_k:P_j (E)	d	0	$3d$	0

Table 3: P_i's timeliness bounds for early ordering m given that MC_i first exceeded $m.TS$ by accepting m'

To discuss the new timeliness bounds in table 3, we will suppose the following: let non-faulty P_i receive m at its local clock time t_i when $MC_i > m.TS$. Let m' be a message which P_i has earlier accepted at local clock time t_i', $t_i' < t_i$, and let $m'.TS \geq m.TS$. Note that, unlike in Section 3.2, m'

need not be the message whose acceptance by P_i first set $MC_i > m.TS$. Let us measure time according to P_i's clock and drop the suffix i from t_i and t_i'. The context that allows the timeliness bound to be reduced to zero is: *m* and *m*' having the same *immediate_sender* when *m* is single-signed, and *m* and *m*' having arrived through the same path when *m* is double-signed. This gives rise to the following two categories which are discussed in turn: (i) *m* is single-signed and the *immediate_sender* of *m* = the *immediate_sender* of *m*', and (ii) *path*(*m*) = *path*(*m*') when *m* is double-signed. For both the categories, we will show that the fifo communication property requires that *m* (the message with smaller timestamp) be received before *m*' (the message with larger timestamp) if the *immediate_sender* of *m* is non-faulty; that is, *m* received after *m*' be regarded as untimely.

Category 4.1: *m* is single-signed and the *immediate_sender* of *m* = the *immediate_sender* of *m*'= P_j (say).

Since *m* is single-signed, P_j is also the *originator* of *m*. In order to estimate the safe bound, we will fix P_j as non-faulty. When P_j either forms and sends or diffuses (double-signed) *m*' to P_i, $MC_j > m'.TS$ is true. Since $m.TS$ is not larger than $m'.TS$, P_j must have formed and sent *m* before it sent *m*' to P_i. Due to fifo communication, P_i must receive *m* before *m*'; that is, $(t_i - t_i') < 0$. This lets the bound of *d* in the entries of B1, D1, C2, and E2 in table 1 be replaced by zero in table 3.

w

Category 4.2: *path*(*m*) = *path*(*m*') and *m* is double-signed.

Since *path*(*m*) = *path*(*m*'), *m*' is also double-signed. Let P_k be the *immediate_sender* of *m* and also the *immediate_sender* of *m*'. Obviously, the *originator* of *m* = the *originator* of *m*' = P_j. Let us fix P_k as non-faulty. Suppose that P_k receives from P_j the single-signed *m*' first and then the single-signed *m*. By the arguments for the previous category, P_k must regard the single-signed *m* as untimely and not diffuse it to P_i. In other words, if non-faulty P_k is to diffuse *m* and *m*' to P_i, then it must receive and accept the single-signed *m* first and then the single-signed *m*'. Messages are diffused in the order they are accepted; hence P_i must receive from P_k first *m* and then *m*'. In other words, if P_i receives *m* after it accepts *m*', P_k must be faulty and *m* should be regarded as untimely. So, in this category also we have $(t_i - t_i') < 0$ and this makes the entries of D3 and E4 in table 3 to become zero. w

These reductions in timeliness bounds result in the following rule which must take priority over other rules where applicable:

Rule 5: Consider a processor that receives a message *msg* after it has accepted *msg*', $msg'.TS \geq msg.TS$. It should consider *msg* as untimely when:

msg is single-signed and has the same *immediate_sender* as *msg*', or *msg* is double-signed and *path*(*msg*) = *path*(*msg*'). m

The zero timeliness bound for the Category 4.2 messages has introduced the following situation which requires the correctness reasoning of Lemma 4 to be revisited: consider that P_i has accepted m' with $path(m') = P_j{:}P_k$ at t'; it can accept the single-signed m from P_j until $t'+d$ as per D2, while the double-signed m with $path = P_j{:}P_k$ received immediately after t' must be regarded as untimely as per D3. Thus, in certain situations, a message with single signature is given a longer acceptance delay than its *equiv* message with two signatures. The zero bound in D3 and E4, and the d bound in D2 and E1 give rise to these situations. In the previous protocol, if P_i receives and accepts a single-signed m at t, then it will certainly find any double-signed m', $m'.TS \geq m.TS$, that it receives before t as timely. This statement is not always true for the early-order protocol and therefore the Lemma 4 in its present form becomes invalid. This, in turn, invalidates some of the arguments used for reducing the timeliness bounds of category 3.3 messages from $4d$ to $3d$. We argue below that the bound of $3d$ is still applicable for messages of that category; first, we modify lemma 4 for the early order version.

Lemma 5: Let t_1 and t_2 be the local clock times when a non-faulty P_i receives a single-signed m_1 and a double-signed m_2, respectively; also, let $m_1.TS \leq m_2.TS$ and $t_1 > t_2$. If P_i finds m_1 timely then it must also find m_2 timely, provided the *immediate_sender* of m_2 is non-faulty.

Proof: As in lemma 4, we will measure time according to P_i's clock and prove the lemma by contradiction. Suppose that P_i finds m_2 late. Let m_2 be late by y, $y > 0$, time units, i.e., if P_i had received m_2 at any time before $(t_2 - y)$ then it would have found m_2 timely. That m_2 received at t_2 was found late implies that there exists a message m', $m'.TS \geq m_2.TS$, which was accepted by P_i at $(t_2 - y - tb_2)$, where tb_2 is the timeliness bound indicated by the entry of Table 3 whose row corresponds to the $path(m')$ and column to the $path(m_2)$. Let tb_1 is the timeliness bound indicated by the entry of Table 3 whose row corresponds to the $path(m')$ and column to the $path(m_1)$.

Given that P_i receives m' before m_2, the arguments provided for reducing the timeliness bounds for the category 4.2 messages indicate that $path(m')$ and $path(m_2)$ cannot be the same if the *immediate_sender* of m_2 is non-faulty. For any given $path(m')$, $path(m') \neq path(m_2)$, the entries of Table 3 indicate that $tb_1 < tb_2$; that is, disregarding the entries D3 and E4, the minimum of column 3 and column 4 entries in every given row is larger than the maximum of column 1 and column 2 entries. The arguments of Lemma 4 can now be used to show that $t_1 > (t_2 - y - tb_2 + tb_1)$ and that m_1 cannot be found timely by P_i. This is a contradiction. q

Category 3.3: the *originator* of m = the *originator* of m' = P_k (say) and $path(m) \neq path(m')$.

Let us suppose that P_j, the *immediate_sender* of m, is non-faulty. Since $path(m) \neq path(m')$ and m is a double-signed message, m' must be a single-signed message. So, P_i diffuses m' which it accepts, to P_j and the diffused message must reach P_j before $t'+d$. P_j can receive the diffused double-signed m' either before or after it receives the single-signed m from P_k. Say, P_j receives the single-signed m from P_k before it receives m' from P_i, i.e., no later than $t'+d$. In that case, P_i must receive the diffused m from P_j no later than $t'+d+d$; that is, $(t - t') < 2d$.

Let us consider the case where P_j receives the single-signed m from P_k after it has received the double-signed m' from P_i. Lemma 5 can now be applied as its premises are met: P_j finds the single-signed m timely, $m.TS \leq m'.TS$, the double-signed m' arrives before m, and the *immediate_sender* of m' is non-faulty. Therefore, P_j must also find the double-signed m' timely. Having found m' timely, P_j cannot apply rule 5 in checking the timeliness of m because the *immediate_sender* of m is not the same as the *immediate_sender* of m'; only rule 4 can be applied. From the arguments provided earlier for this message category, it follows that P_i could receive the diffused m from P_j before $t'+d+d+d$. Thus, $(t - t') < 3d$ is possible and $3d$ is still the timeliness bound for messages of this category. w

4.2. Protocol Description and Correctness

The early-order protocol is different from the protocol of figure 2 only in the following three aspects: (i) some of the timeliness bounds have been safely reduced to zero from d; (ii) generation and transmission of null messages; and, (iii) filtering null messages from the *stable* list and ordering only non-null messages. The differences due to (i) and (ii) lead to changes in the *Diffuse* process and (iii) is handled in the *Order* process; the *Broadcast* process remains the same as before and is omitted in the following presentation of the early-order protocol.

```
do /* Broadcast process handling messages received from outside the node */
  ... /* same as the protocol of Section 3 */
end do    /* Broadcast process */
||
do /* Diffuse process handling messages received from internal links */
  receive(m);
  if (m.TS ≤ PCi[path(m)] then discard(m)
  else
    MCi = max{MCi, m.TS+1};
    t = clocki;
    case path(m) == Pk
      PCi[Pk] = m.TS;                              /* implements B1 */
      schedule update(PCi[Pj], m.TS) at t+2d;      /* implements B2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+3d;   /* implements B3 */
      schedule update(PCi[Pk:Pj], m.TS) at t+3d;   /* implements B4 */
    case path(m) == Pj
      schedule update(PCi[Pk], m.TS) at t+2d;                    /* C1 */
      PCi[Pj] = m.TS;                                            /* C2 */
      schedule update(PCi[Pj:Pk], m.TS) at t+3d;                 /* C3 */
      schedule update(PCi[Pk:Pj], m.TS) at t+3d;                 /* C4 */
    case path(m) == Pj:Pk
      PCi[Pk] = m.TS;                                            /* D1 */
      schedule update(PCi[Pj], m.TS) at t+d;                     /* D2 */
```

```
        PCi[Pj:Pk], m.TS;                                             /* D3 */
        schedule update(PCi[Pk:Pj], m.TS) at t+3d;                    /* D4 */
      case path(m) == Pk:Pj
        schedule update(PCi[Pk], m.TS) at t+d;                        /* E1 */
        PCi[Pj] = m.TS;                                               /* E2 */
        schedule update(PCi[Pj:Pk], m.TS) at t+3d;                    /* E3 */
        PCi[Pk:Pj] = m.TS;                                            /* E4 */
      end case
      send(m);
      acceptedi = acceptedi ∪ {m}
      if m.μ ≠ ϕ then                                  /* if the accepted m is non-null...*/
        msg.μ = ϕ; msg.TS = MCi; msg.O = i;            /* form and send a null msg */
        MCi = MCi+1;
        send(msg);
      end if
    end if
  end do    /* Diffuse process */
  ||
  do  /* Order process identifying and ordering stable non-null messages  */
    PCi,mn = min{PCi[p] | ∀ p: p ∈ {Pk, Pj, Pj:Pk, Pk:Pj}}
    while SCi < PCi,mn  do
      SCi = SCi + 1;
      stablei = {m | m ∈ acceptedi and m.TS = SCi};
      acceptedi = acceptedi-stablei;
      if stablei ≠ {} then
        distinct = { };
        for every m in stablei:
          if (equiv(m) ∉ distinct and m.μ ≠ ϕ) then insert m in distinct end if
        end for
        stablei = distinct;
        spurious = {m, m' | m∈ stablei ∧ m'∈ stablei ∧ m'.O=m.O ∧ m'.μ≠m.μ};
        stablei = stablei-spurious;
          if stablei ≠ { } then order and deliver each m in stablei according to « relation end if;
      end if
    end while
  end do    /* Order process */
```

Figure 3. The early-order Protocol.

As in the previous protocol, an efficient implementation of the above protocol can avoid entering an accepted non-null *m* into the *accepted* list if an *equiv*(*m*) is already there. Further, forming and sending of a null message can also be avoided if the received non-null message is not to be entered into the *accepted* list.

Theorem 2: Given that $d \geq \delta/(1-5\rho)$, the early-order protocol guarantees *unanimity* and *validity* conditions with $\Delta = 4d(1+\rho)$, and with $\Delta = 3\delta$ in the absence of failures.

Proof: Since the bounds of table 3 are proved in the appendix to be safe for $d \geq \delta/(1-5\rho)$, the arguments of theorem 1 can be used to conclude that the *unanimity* and *validity* conditions are satisfied for $\Delta = 4d(1+\rho)$. To see how early ordering is achieved in the absence of failures, suppose that all processors are non-faulty. Let P_i form and send *m* at real time T_i. P_j and P_k

receive and accept m not later than $T_i+\delta$. Upon accepting m, P_j and P_k form and send null messages, say, m_j and m_k respectively, such that $m_j.TS > m.TS$ and $m_k.TS > m.TS$. P_i accepts these messages before $T_i+2\delta$ and sets $PC_i[P_j] = m_j.TS > m.TS$ and $PC_i[P_k] = m_k.TS > m.TS$ due to the instructions implementing C2 and B1 respectively. P_j and P_k respectively diffuse to P_i the double-signed, null messages m_k and m_j which P_i will accept before $T_i+3\delta$ and will subsequently set $PC_i[P_k:P_j] = m_k.TS > m.TS$ and $PC_i[P_j:P_k] = m_j.TS > m.TS$ due to the instructions implementing E4 and D3. Thus, before $T_i+3\delta$, $PC_{i,mn}$ becomes larger than $m.TS$ and m gets ordered by P_i.

Let us consider the ordering of m by P_k. Let us imagine that the suffixes of i and k are interchanged in the above presentation of the protocol, to help study the behaviour of P_k during protocol execution. P_k accepts the single-signed m before $T_i+\delta$, and sets $PC_k[P_i] = m.TS$ due to the instruction implementing B1. P_k accepting the single-signed m_j before $T_i+2\delta$ and accepting the double-signed m_j before $T_i+3\delta$ set $PC_k[P_j] = m_j.TS > m.TS$ and $PC_k[P_j:P_i] = m_k.TS > m.TS$ due to the instructions implementing C2 and D3 respectively. P_k accepts the double-signed m from P_j before $T_i+2\delta$ and sets $PC_k[P_i:P_j] = m.TS$ due to the instructions implementing E4. So, $PC_{k,mn}$ becomes $m.TS$ before $T_i+3\delta$, leading to ordering of m by P_k. So a message gets ordered by all processors within 3δ time of it being formed and sent, if all processors are non-faulty. q

5. Implementation and Performance Measurements

We have implemented three different versions of a TMR node on a network of T800 Inmos transputers [INMOS88], each of them incorporating a distinct order protocol for the ordering of input messages. Sets of experiments have been carried out on the various node versions implemented, and the performance in each case has been measured. In this section we describe the node versions implemented and illustrate their relative performance; we also compare the performance figures obtained against the expected values and thus validate our measurements. A detailed description of the implementation platform can be seen in [Brasil95a]. (The experiments and the results reported here are more elaborate and extensive than the ones appeared in [Brasil95b].)

All the TMR versions implemented incorporate a majority voting protocol and output only a voted message which can be verified as such by any recipient of the message (see [Shri92, Speirs93] for details). The versions are named after the protocols they employ for ordering input messages:

i) TMR synchronised clocks: this version employs the Byzantine resilient, synchronised-clock based protocol of [Cri85];

ii) *TMR logical clocks*: this version uses the protocol presented in Section 3; and

iii) *TMR early-order*: in this version, the early-order protocol of Section 4 is used. (Communication between processors forming this node satisfy the *fifo* property, which is easily achieved within the transputer based architecture used.)

The general structure of the experiments carried out is that of a client application process executing on an unreplicated node and making requests to a server application process which executes on a TMR node. We are particularly interested in the overheads associated with ordering of input messages and voting of output messages. Thus, we have measured the following time intervals for each TMR node version:

Input delay (ID): This measures the time interval between a client message being received by at least one non-faulty processor of the node and the message being removed for processing by a majority of the processors of the node. Thus, the input delay is made up of the ordering delay for a message plus any overheads associated with authentication and queue manipulation within the node.

Output delay (OD): The output delay measures the time interval between a message becoming ready for voting at a majority of processors, and the voted message being output by at least one non-faulty processor.

Node delay (ND): the node delay is simply the sum of the input and output delays (ID+OD). It represents the overhead due to replication.

From the perspective of the client node we have measured the following time interval:

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

5.1. Overhead for a simple client-server application

In the first set of experiments the client process executing on an unreplicated node requests a simple service from the server application process executing on a TMR node. The client process broadcasts a request to each server replica that is running on a distinct processor of the TMR node. It then waits for a response from any of the server replicas. Each server replica services the ordered request and majority-votes the response it produces. The client issues a new request after receiving a voted response message from a server replica.

We have collected data for ten runs of the experiment, each run differs in service times and involves the client node sending 100 request messages of 64 bytes and receiving response messages of approximately the same size. For each run we have measured ID, OD, ND, and RL.

We have also measured the average link transmission delay of internal messages, making a distinction between the transmission delay of internal messages associated with the ordering of input messages (δ_{input}), and the transmission delay of internal messages associated with the majority-voting of response messages (δ_{output}). Table 4 gives the average delays for different TMR node implementations.

TMR Node Version	ID (ms)	OD (ms)	ND (ms)	δ_{input} (ms)	δ_{output} (ms)
TMR synchronised clocks	202.83	10.45	213.28	5.44	4.27
TMR logical clocks	136.28	10.20	146.48	9.18	4.09
TMR early-order	35.96	10.31	46.27	9.70	4.30

Table 4: Performance overhead for a client-server application on TMR nodes

We have exposed the implementations to worst case situations and measured the smallest safe value for δ, which turned out to be δ = 50 ms. Also, for the synchronised clock based implementation we have used the fault-tolerant clock synchronisation algorithm presented in [Halp84], for which we can choose the maximum error on the reading of the clocks of any two non-faulty processors to be $e = \delta/(1+\rho) \approx 50$ ms, which leads to $\Delta' = 2(d+e) \approx 200$ ms. We can deduce from the value of ID presented in Table 4 that the authentication and queue manipulation activities in the synchronised clock based TMR node consume approximately 2.83 ms.

To validate our implementation of the logical clock TMR node we will approximately estimate the ordering delays for that node and verify whether the measured delays agree with the estimated value. Recall that the client process sends a copy of its request to all processors in the TMR node. Let T_1, T_2 and T_3, $T_1 \le T_2 \le T_3$, be the realtime instances when the replicas of a given request, say, R is received by the processors. If one of the processors within the node fails to receive the request message, T_3 becomes infinity; if the client is to send its request only to two processors, T_2 may also become infinity. Let us define $\lambda_2 = T_2\text{-}T_1$, and $\lambda_3 = T_3\text{-}T_1$. Let processor P receive the request at T_1; it will order the message it sends with $m.\mu = R$ approximately at T_1+4δ (assuming ρ of P's clock is zero and $d = \delta$). If λ_2 is finite, P will receive a single-signed message with $m.\mu = R$ approximately at $T_1+\lambda_2+\delta_{input}$ which becomes stable at $T_1+\lambda_2+\delta_{input}+3\delta$. If λ_3 is also finite, each processor will receive two double-signed messages with $m.\mu = R$ at or before $T_1+\lambda_3+2\delta_{input}$ which become stable no later than $T_1+\lambda_3+2\delta_{input}+2\delta$. Thus, the Δ of the logical clock TMR node is approximately the minimum of 4δ, $\lambda_2+\delta_{input}+3\delta$ and $\lambda_3+2\delta_{input}+2\delta$. In our experiments, the average values of λ_2 and λ_3 turned out to be 2.04 ms and 13.15, respectively; thus, the average ordering delay $= \lambda_3+2\delta_{input}+2\delta$, and from the value of ID shown in Table 4 we see that this figure is realised in practice. (Approximately 4.77 ms are spent for message authentication, queue manipulation and scheduling of *update* processes.)

For early-order TMR node, the average value expected for its ordering delay is given by $3\delta_{input}$. Thus, from the value of ID presented in Table 4 we can see again that the measured values are in close agreement with the theoretical estimations. The extra overhead (6.75ms) can be attributed to: (i) the overheads associated with message authentication, queue manipulation and scheduling of *update* processes; (ii) the overheads associated with the management of null acknowledgement broadcasts; and (iii) the event triggered nature of the early-order protocol which is responsible for extra overheads in the scheduling of processes.

The overheads for message comparison during voting is (OD-δ_{output}) and is pretty much the same for all three implementations (6.18 ms, 6.11 ms, and 6.01 ms for the synchronised clock based, logical clock based, and early-order implementations, respectively) as we would expect.

Now we analyse how the overheads at the node where the server process is executed are perceived by the client process executing on the unreplicated node. We have modified the server process of the previous experiment in such a way that we can control its processing time. We then executed the experiment for different server processing times, and measured the average response latency (RL) experienced by the client process executing on the unreplicated node when the client issues requests to server processes executing on the different TMR nodes implemented. We have also measured the response latency for the case where the server process executes on an unreplicated node ($RL_{unreplicated}$). We define the Relative Replication Performance Overhead (RRPO) of a TMR node as: RRPO = 1-$RL_{unreplicated}$/RL. Figure 4 shows the RRPO obtained for the various TMR nodes implemented, when the server processing time was increased from (almost) zero up to 100 ms.

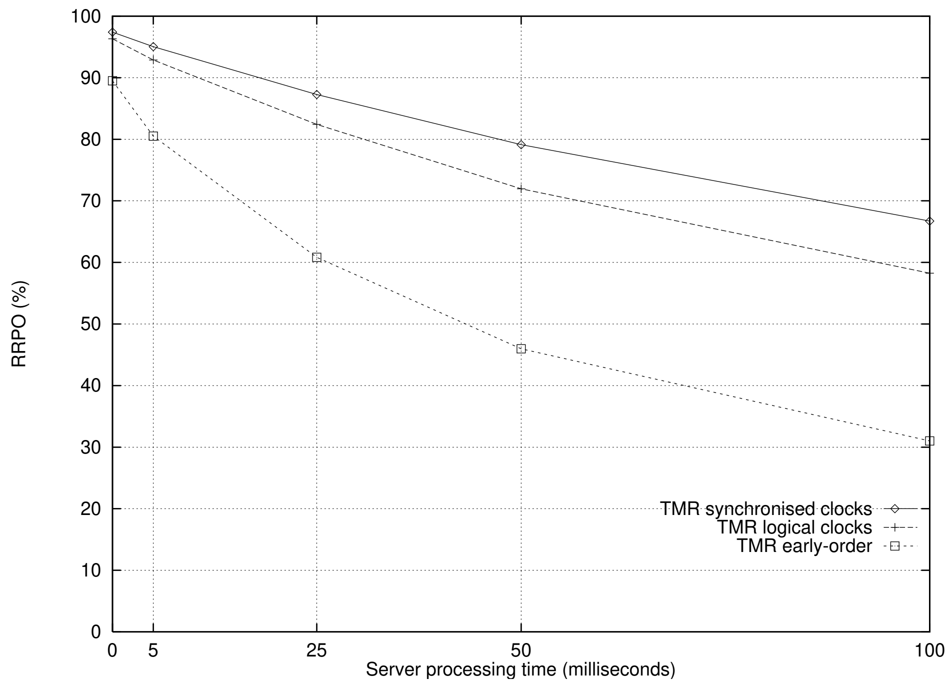

Figure 4: RRPO versus server processing time

As expected, when the server processing time increases there is a reduction on the RRPO of TMR nodes. The curves presented in Figure 4 show that the price in performance becomes significant only in those applications where processes interact frequently. If on the other hand, application processes are involved in computations requiring less frequent interactions, then the performance impact of replication protocols can be quite small. For instance, the RRPO of the early-order TMR node is reduced from more than 80% to nearly 30% when the server processing time is increased from almost zero to 100 ms.

5.2. Node delay versus message size

The next experiment was performed to evaluate the impact of the size of messages on the performance of a TMR node. Transputers use a byte-stream protocol for link-level communication. In our system, the end-to-end message transmission delay between two transputers varied from 1.45 ms (messages of size 256 bytes) to 2.16 ms (messages of size 1536 bytes). Thus, the size of messages will affect intra-node message transmission times, consequently affecting both input and output delays.

Using the original client-server application previously described, we have measured the node delay for the various TMR nodes implemented, as the message size was increased from 256 to 1536 bytes. Figure 5 presents the average node delays obtained.

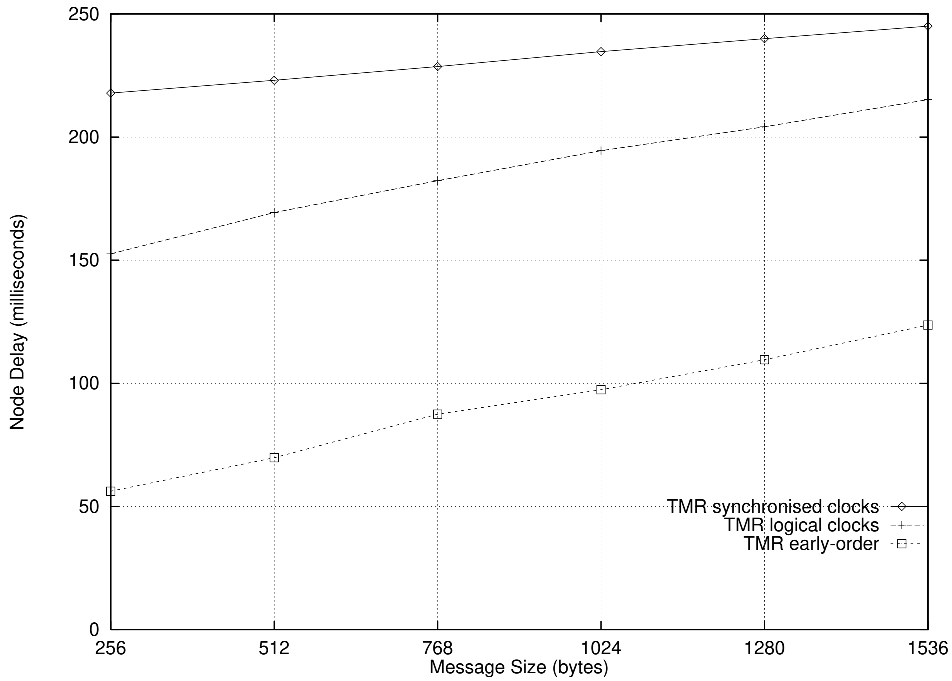

Figure 5: Node delay versus message size

Increasing the message size impacts the various node delays in a diverse fashion. The large internal message traffic, particularly during the input ordering phase, is responsible for a sharp increase in the node delay of the early-order implementation (52.69 μs/byte). On the other hand, the synchronised clock based implementation suffers only a relatively small increase of 21.21 μs/byte. This is because the increased transmission delay will have no impact on the performance of the synchronised-time based order protocol, since its ordering delay is based on the worst case transmission delay. Finally, the node delay of the logical clock based implementation suffers an intermediary increase of 48.93 μs/byte. This is because, a component of the ordering delay of this implementation is a function of the constant, worst case transmission delay (δ), and the other component is a function of the actual transmission delays (δ_{input}) which increases with the message size.

5.3. Overhead when one processor in the node is faulty

In our last set of experiments we have assessed how the failure of one of the processors of a TMR node affects the overhead. We have arbitrarily chosen one of the processors of the node and we have deliberately injected faults in this processor. We have then executed our original RPC-like experiment and have measured the various node delays. Table 5 summarises the figures obtained when we simulate a crash failure of one of the processors forming the node. The crash scenario is achieved by injecting omission faults on both internal links of the processor (see [Tao95] for more details on this fault injection technique).

TMR Node Version	ID (ms)	OD (ms)	ND (ms)
TMR synchronised clocks	202.75	8.28	211.03
TMR logical clocks	163.88	8.73	172.61
TMR early-order	162.57	8.01	170.58

Table 5: Performance overhead for TMR nodes containing a crashed processor

As expected, the performance of the early-order node implementation is the most affected by the failure of one of the processors of the TMR node. The sharp increase of the node delay of the early-order implementation is due to the fact that its early-order protocol only achieves optimum performance when the node is in a failure-free state. As indicated by the values shown in Table 5, when there is a faulty processor within the node the early-order protocol has its performance comparable to the performance of the protocol presented in Section 3. The logical clock based implementation also suffers an increase on its ID. This is because for this experiment we have $\lambda_3 = \infty$, and therefore the average stability delay is given by $\lambda_2+\delta_{input}+3\delta$.

On the other hand, for the synchronised clock based implementation, the failure of one processor does not cause an increase on its node delays. In fact, for crash failures such as the one we have simulated, the node delay of the synchronised clock version is slightly reduced. This reduction is due to the fact that since one processor has crashed, the internal traffic of messages is reduced, which in turn decreases the internal message transmission delays. Comparing the values shown in Table 5 with those presented in Table 4, we can see that the output delay (OD) of all three implementations is reduced by about 2 ms when there is a crashed processor within the node.

Most of the performance overheads of the TMR node implementations are associated with the ordering of input messages. In a lightly loaded, error-free system, the early-order implementation presents a much reduced performance overhead when compared with the performance of both the synchronised clock and the logical clock based TMR implementations. For our particular system, the early-order implementation out-performs the synchronised clock based implementation by a factor of 4.61, and the logical clock based implementation by a factor of 3.17. On the other hand, when there is a faulty processor within the node the performance of the early-order implementation is comparable to that of the logical clock based implementation, still out-performing the synchronised clock based implementation by a factor of 1.24.

6. Concluding Remarks

Through our research partnership work with industries [Speirs97], we have learnt that the software implementation of replicated processing is increasingly being preferred to the hardware approach for many (particularly Telecom) applications. The main reason for this preference is the ease with which the advances in processor technology can be accommodated. Since the principles behind replica management does not change with the underlying hardware, software can be easily ported to any type of processors, including the ones expected to be available in future. In this paper, we have presented two message ordering protocols for software implementation of TMR systems. They use timeouts and logical clocks, obviating the need to explicitly synchronise processors' clocks within some known bound. The second protocol is designed to provide fast message ordering in the absence of faults. We have built TMR nodes using these protocols and a representative, time based ordering protocol. The performance measurements indicate that our protocols perform faster than the time based one, both in the presence and absence of failures; and also that they can considerably benefit from a situation where processors receive a given input replica at times that are closer to each other, i.e., λ_2 and λ_3 are small.

We conclude our paper by addressing two features of our protocols. First, can they be generalised to a π-out-of-N system, where at most π out of N processors can fail? They can be, but the resulting protocols will not be faster than their time based counterparts. In a TMR system, a processor that receives a double-signed message knows that every other processor has already seen that message. In a π-out-of-N system, however, a processor that receives a $(\pi+1)$-signed message can only know that every other processor in the system would see that message within *d* time after it has received the message. This extra *d* makes the ordering delay to be the same as the worst case ordering delays of time based ordering protocols. Also, the number of path counters to be maintained increases exponentially with π and N: one counter for every possible path through which a message with at most $(\pi+1)$ signatures can be received. These drawbacks must be borne in mind when the protocols presented here are extended for a system of more than three processors. Secondly, can there be a timeout based protocol for a TMR system that guarantees a smaller ordering delay than the first protocol? It appears not. The timeliness bound reduction rules of sub-section 3.3 indicate that Δ cannot be smaller than $4d$ in all circumstances, and the lemmas of appendix show that d must be at least $\delta/(1-5\rho)$. So, there are reasons to believe that the protocols presented here are the optimal, timeout based message ordering protocols for a TMR system.

Acknowledgements

This work has been supported in part by grants from CNPq/Brazil, the EPSRC grant No. GR/H47999 and the Hewlett-Packard Research Partnership Grant. Comments from Santosh Shrivastava were helpful.

References

[Brasil95a] F.V. Brasileiro, “Constructing Fail-Controlled Nodes for Distributed Systems: A Software Approach," Ph.D. Thesis, University of Newcastle upon Tyne, May 1995.

[Brasil95b] F V Brasileiro, P D Ezhilchelvan and N A Speirs, "TMR processing without explicit clock synchronisation", The Proceedings of the 14th Symposium on Reliable Distributed Systems, Bad Neuenahr, Germany, September 1995, pp. 186-195.

[Brasil96] F V Brasileiro, P D Ezhilchelvan, S.K. Shrivastava, N A Speirs, and S. Tao " Implementing Fail-Silent nodes for Distributed Systems", IEEE Transactions on Computers, 45 (11), pp. 1226-1238, Nov. 1996.

[Cri85] F. Cristian, et. al., “Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement," Digest of Papers, FTCS-15, Ann Arbor, USA, pp. 200-206, June 1985.

[Dolev82] D. Dolev and H.R. Strong, “Requirements for Agreement in a Distributed System," Proc. 2nd Symposium on Distributed Databases, pp. 115-129, September 1982.

[Dolev84] D. Dolev et. al., “On the Possibility and Impossibility of Achieving Clock Synchronisation," Proc. 16th Annual ACM STOC, pp. 504-511, April 1984.

[Fisch85] M.J. Fischer et. al., “Impossibility of Distributed Consensus with one faulty Process," JACM 32(2), pp. 374-382, 1985.

[Halp84] J.Y. Halpern et. al., “Fault-tolerant Clock Synchronisation," Proceedings of the 3rd ACM symposium on Principles of Distributed Computing, pp. 89-102, August 1984.

[INMOS88] INMOS Limited, Transputer Instruction Set, Prentice Hall International (UK) Ltd, 1988, ISBN 0-13-929100-8.

[Kopetz87] H.Kopetz and W. Ochsenreiter, "Clock Synchronisation in Distributed real time systems", IEEE Transactions on Computers, Vol C-36 (8), pp. 933-940, 1987.

[Lampo78] Lamport, L., "Time, Clocks, and Ordering of Events in a Distributed System", Commun. of ACM, 21, 7, July 1978, pp. 558-565.

[Lampo84] L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems", ACM Transactions on Programming Languages and Systems, 6 (2) pp. 254-280, April 1984.

[Lampo85] L. Lamport and P.M. Melliar-Smith, “Synchronising Clocks in the Presence of Faults," JACM 32(1), pp. 52-78, January 1985.

[Pease80] M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the Presence of Faults," Journal of the ACM, Vol. 27(2), pp. 228-234, April 1980.

[Rivest78] R.L. Rivest, et. al., “A method for obtaining Digital Signatures and Public Key Cryptosystems," CACM 31(2), pp. 120-126, February 1978.

[Schmuck90] F. Schmuck and F. Cristian, “Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronisation Algorithm," Proceedings of the 9th ACM symposium on Principles of Distributed Computing, pp. 133-141, August 1990.

[Sch84] F.B. Schneider, “Byzantine Generals in Action: Implementing Fail-Stop Processors," ACM Transactions on Computer Systems, Vol. 2(2), pp. 145-154, May 1984.

[Sch90] F.B. Schneider, “Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial," ACM Computing Surveys, Vol. 22(4), pp. 299-319, December 1990.

[Shri92] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, “Principle Features of the Voltan Family of Reliable Node Architectures for Distributed Systems," IEEE Transactions on Computers, 41(5), pp.542-549, May 1992.

[Speirs93] 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. 1-17, November 1993.

[Speirs97] N.A. Speirs, S.K. Shrivastava, P D Ezhilchelvan, F. Brasileiro and S. Tao, "Efficient schemes for constructing fail-silent nodes in distributed systems", The UK Patent No. 2288045, 1997. (The US Patent Application in progress.)

[Srikanth85] T.K. Srikanth and S. Toueg, “Optimal Clock Synchronisation," Proceedings of the 4th ACM symposium on Principles of Distributed Computing, pp. 71-86, August 1985.

[Tao95] S. Tao, P.D. Ezhilchelvan and S.K. Shrivastava, “Using Focused Fault Injection Testing for Validating Software Implemented Fault Tolerance Mechanisms," Distributed Systems Engineering Journal, Vol 2 (1), pp. 39-49, March 95.

[Wensl78] J.H. Wensley et. al., “SIFT: Design and Analysis of a Fault Tolerant Computer For Aircraft Control," Proceedings of IEEE, 66 (10), pp. 1240-1255, October 1978.

Appendix

Compensating for clock drift

We prove that setting $d \geq \delta/(1\text{-}5\rho)$ compensates for the differences caused by the non-identical running rates of non-faulty processors' clocks, and thus extends the safety property of the timeliness bounds identified earlier into a system context where the running rates of non-faulty clocks are within a known bound ρ. Specifically, we prove that the following is guaranteed in any execution of our protocols if $d \geq \delta/(1\text{-}5\rho)$: when a non-faulty P_k receives a single- or double-signed message m from another non-faulty P_i, P_k finds m timely.

Before presenting the proofs, we will recap some relevant points and assumptions stated earlier and introduce some new notation. We assume that a non-faulty processor executes the instructions of the protocols in zero real/clock time; we adopt the style of writing real time values in Greek or italicised upper case Roman letters, and clock time values in italicised lower case Roman letters; the term 'clock' is used to refer to a processor's physical clock. We will also ignore the second and higher order powers of ρ in our proofs. The following two assumptions of section 2 and lemma 1 are relevant here.

Assumption 4: If a non-faulty P_i prepares and transmits m at real time T_i, every non-faulty destination P_k will receive m at real time T_k, $T_i \leq T_k < (T_i+\delta)$, where δ, $\delta > 0$, is known.

Assumption 5: A non-faulty P_i's clock measures a clock time interval x in a real time interval $x(1+\rho_i)$, where $|\rho_i| \leq \rho$ and ρ is a known positive constant. That is, ρ_i is the rate at which the clock of processor P_i drifts from real time.

Lemma 1: If a non-faulty P_i accepts m at real time T_i, then any other non-faulty P_k either sends or receives an authentic *equiv*(m) at real time T_k s.t. $|T_i\text{-}T_k| < \delta$.

We will use the following new notations: for path p, $START_i(p, \geq ts)$ denotes the smallest real time instance when $PC_i[p]$ starts being larger than or equal to ts, $ts \geq 1$. That is, just before $START_i(p, \geq ts)$, $PC_i[p]$ is less than ts and from $START_i(p, \geq ts)$ onwards, any received m with $path(m) = p$ and $m.TS \leq ts$, is regarded by P_i as untimely. Further, $END_i(\leq ts)$ denotes the largest real time instance when MC_i ends being less than or equal to ts. That is, at $END_i(\leq ts)$ P_i accepts a message with timestamp ts; immediately after $END_i(\leq ts)$, MC_i becomes larger than ts and P_i would no longer form and send a message m with $m.TS \leq ts$.

Lemma S1: For $d \geq \delta/(1\text{-}\rho)$ and for non-faulty P_i and P_k, $START_i(P_k, \geq ts)\text{-}END_k(\leq ts) \geq \delta$ is true in any execution of the first protocol.

Proof: Let the execution of the *update* process that sets $PC_i[P_k] \geq ts$ at $START_i(P_k, \geq ts)$ be due to P_i accepting m', $m'.TS \geq ts$, say, at real time $TIME_i$. We will consider two cases, depending on whether P_k has signed m':

(i) P_k is the *originator* or the *immediate_sender* of m'; MC_k is ensured to be larger than $m'.TS$ when P_k accepts m' which must be at or before $TIME_i$. Therefore, $END_k(\leq ts) \leq TIME_i$. The entries B1, D1 and E1 of Table 1 indicate that $START_i(P_k, \geq ts) = TIME_i + d(1+\rho_i)$. Thus, $START_i(P_k, \geq ts) \geq END_k(\leq ts) + d(1+\rho_i)$; since $d(1+\rho_i) \geq \delta$ is true when $d \geq \delta/(1-\rho)$, we have $START_i(P_k, \geq ts) - END_k(\leq ts) \geq \delta$.

(ii) P_k is neither the *originator* nor the *immediate_sender* of m'; Since P_i accepts m' at $TIME_i$, by lemma 1, P_k receives an *equiv*(m'), say m_k, at some real time $TIME_k$, $TIME_k < TIME_i + \delta$. If m_k is found untimely, then $PC_k[path(m_k)]$ and therefore MC_k must be larger than ts when m_k was received. (Note: MC_k is always ahead of any path counter of P_k.) So, $TIME_k > END_k(\leq ts)$. If, on the other hand, m_k is found timely, then MC_k is set to be larger than $m'.TS$ at $TIME_k$, if MC_k is not already larger than $m'.TS$. Since $m'.TS \geq ts$, $TIME_k \geq END_k(\leq ts)$. In both cases, we have $END_k(\leq ts) \leq TIME_k < TIME_i + \delta$. When m' is not signed by P_k, the entries A1 and C1 indicate that $START_i(P_k, \geq ts) = TIME_i + 2d(1+\rho_i)$. Therefore, $START_i(P_k, \geq ts) > END_k(\leq ts) - \delta + 2d(1+\rho_i)$. $d \geq \delta/(1-\rho)$ implies that $2d(1+\rho_i) \geq 2\delta$. So, $START_i(P_k, \geq ts) - END_k(\leq ts) > \delta$. Hence the lemma. q

Lemma S2: For $d \geq \delta/(1-5\rho)$ and for non-faulty P_i and P_k, $START_k(P_j{:}P_i, \geq ts) - START_i(P_j, \geq ts) \geq \delta$ is true in any execution of the first protocol.

Proof: Let us abbreviate $START_k(P_j{:}P_i, \geq ts)$ and $START_i(P_j, \geq ts)$ as $START_k(P_j{:}P_i)$ and $START_i(P_j)$, respectively. Let m', $m'.TS \geq ts$, be the message whose acceptance by P_k scheduled the execution of the *update* process that sets $PC_k[P_j{:}P_i] \geq ts$ at $START_k(P_j{:}P_i)$. Regarding the *originator* of m', we consider three cases, one for each processor in the node:

(i) the *originator* of $m' = P_k$. To study the timing behaviour of P_k from Table 1, the suffixes i and k there should be interchanged. With this interchange, the entry A3 indicates that P_k has formed and sent m' at real time $START_k(P_j{:}P_i) - 4d(1+\rho_k)$. By assumption 4, P_i receives m' from P_k before $START_k(P_j{:}P_i) - 4d(1+\rho_k) + \delta$. From lemma S1, P_i finds the received m' timely and accepts it. Entry B2 of Table 1 indicates that $START_i(P_j) < START_k(P_j{:}P_i) - 4d(1+\rho_k) + \delta + 2d(1+\rho_i)$. By simple algebra, it can be shown that $START_k(P_j{:}P_i) - START_i(P_j) > \delta$ if $d \geq \delta/(1+2\rho_k-\rho_i)$, which is true when $d \geq \delta/(1-5\rho)$.

(ii) the *originator* of $m' = P_i$. Let m' be received by P_k at $TIME_k$, and be formed and sent by P_i at $TIME_i$, $TIME_i \leq TIME_k$. (Note that m' that P_k receives can be single- or double-signed.)

With suffixes k and i interchanged in Table 1, the entries E3 and B3 indicate that $START_k(P_j{:}P_i) = TIME_k + 3d(1+\rho_k)$; so, $TIME_k = START_k(P_j{:}P_i) - 3d(1+\rho_k)$. For P_i, the entry A2 indicates that $START_i(P_j) \le TIME_i+2d(1+\rho_i)$. Since $TIME_k \ge TIME_i$, $TIME_k \ge START_i(P_j) - 2d(1+\rho_i)$. This means that $START_k(P_j{:}P_i)-3d(1+\rho_k) \ge START_i(P_j)-2d(1+\rho_i)$, and $START_k(P_j{:}P_i)-START_i(1) \ge \delta$ whenever $d \ge \delta/(1+3\rho_k-2\rho_i)$, which is true when $d \ge \delta/(1-5\rho)$.

(iii) the *originator* of $m' = P_j$. With suffixes k and i interchanged in Table 1, the entries D3 and C3 indicate that P_k must have received m' at real time $START_k(P_j{:}P_i)-2d(1+\rho_k)$ or $START_k(P_j{:}P_i)-3d(1+\rho_k)$ depending on whether P_i has signed m' or not, respectively. Let us suppose that P_i has signed the m' which P_k has accepted. Even if m' has taken zero time to reach P_k from P_i, $START_i(P_j) \le START_k(P_j{:}P_i)-2d(1+\rho_k) + d(1+\rho_i)$ as per the entry C2 of Table 1. Simple algebra indicates that $START_k(P_j{:}P_i)-START_i(P_j) \ge \delta$ whenever $d \ge \delta/(1+2\rho_k-\rho_i)$, which is always true when $d \ge \delta/(1-5\rho)$.

Let us now consider the case that P_i has not signed the m' which P_k has accepted. Let P_i receive *equiv*(m') at real time $TIME_i$. By lemma 1, $TIME_i < START_k(P_j{:}P_i)-3d(1+\rho_k)+\delta$. Note that since P_j need not be non-faulty, lemma S1 cannot be used to conclude that P_i will find the single-signed message it might receive directly from P_j, timely. We will consider two possibilities: $TIME_i \ge START_i(P_j)$ or $TIME_i < START_i(P_j)$. If $TIME_i \ge START_i(P_j)$, then we have $START_k(P_j{:}P_i)-START_i(P_j) > \delta$, provided $d \ge (2/3)(\delta/(1+\rho_k))$. If $TIME_i < START_i(P_j)$, P_i finds the received *equiv*(m') timely, whether *equiv*(m') is single- or double-signed. The entries C2 and D2 indicate that $START_i(P_j) \le TIME_i+d(1+\rho_i)$. Thus, $START_k(P_j{:}P_i)-START_i(P_j) > 3d(1+\rho_k)-\delta-d(1+\rho_i)$, and $START_k(P_j{:}P_i)-START_i(P_j) > \delta$ whenever $d \ge \delta/(1+3\rho_k/2-\rho_i/2)$. $d \ge \delta/(1-5\rho)$ implies that $d >$ maximum of $\{(2/3)(\delta/(1+\rho_k)), \delta/(1+3\rho_k/2-\rho_i/2)\}$. q

Case (ii) yields the largest lower bound for d and corresponds to the scenario described at the end of sub-section 3.1.2. Lemmas S1 and S2 prove that the timeliness bounds of Table 1 with $d \ge \delta/(1-5\rho)$ are safe. We will next prove the same for the timeliness bounds used in the early-order protocol.

Lemma S3: Suppose that non-faulty P_i forms and sends m at real-time T and another non-faulty P_k receives m at $T+\delta_m$, where $0 \le \delta_m < \delta$. For $d \ge \delta/(1-\rho)$, $START_k(P_i, \ge m.TS) - T \ge \delta_m$ is true in any execution of the early-order protocol.

Proof: Let the execution of the *update* process that sets $PC_k[P_i] \ge m.TS$ at $START_k(P_i, \ge m.TS)$ be due to P_k accepting m', $m'.TS \ge m.TS$, say, at real-time T'. Let us abbreviate $START_k(P_i, \ge m.TS)$ simply as $START_k(P_i)$. The lemma is trivially true if $m' \equiv m$. Let us suppose that m and m' are not identical. (They may however be *equiv* messages: $m' = equiv(m)$ and $m' \neq m$.) If the *immediate_sender* of m' is P_i, then $T' > T + \delta_m$ because (a) $m'.TS$ must be larger than $m.TS$ and P_i must diffuse m' after it sends m; and (b) P_k receives messages from P_i in fifo order. So, the lemma has to be proved only for the case where the *immediate_sender* of m' is not P_i. This leads to having to consider two cases: (i) P_j is the *immediate_sender* of m' and P_i is the *originator* of m'; and, (ii) m' is single-signed and the *originator* of m' is either P_j or P_k.

(i) P_j is the *immediate_sender* of m' and P_i is the *originator* of m': Since P_i is the *originator* for both m and m', either m and m' are *equiv* messages or $m'.TS > m.TS$ and m' was sent after T. Thus, $T' \geq T$. With the suffixes k and i interchanged in Table 3, the entry E1 indicates that $START_k(P_i) = T' + d(1+\rho_k)$. So, $START_k(P_i) - T \geq d(1+\rho_k)$. When $d \geq \delta/(1-\rho)$, $d(1+\rho_k) \geq \delta > \delta_m$. So the lemma is proved in this case.

(ii) m' is single-signed and the *originator* of m' is either P_j or P_k: with the suffixes k and i interchanged in Table 3, the entries C1 and A1 indicate that $START_k(P_i) = T' + 2d(1+\rho_k)$. The message m' diffused by P_k is received by P_i before $T'+\delta$. P_i can find the diffused m' either timely or untimely. So, MC_i is either ensured to exceed or already larger than $m'.TS$ when m' is received. This means that P_i must have sent m before it receives m', i.e., before $T'+\delta$. So, $T < T'+\delta$. Substituting $T-\delta$ for T' in the equation $START_k(P_i) = T' + 2d(1+\rho_k)$ gives: $START_k(P_i) > T - \delta + 2d(1+\rho_k)$. $START_k(P_i) > T + \delta$ if $d(1+\rho_k) \geq \delta$ which is true when $d \geq \delta/(1-\rho)$. Hence the lemma is proved. q

Lemma S4: Suppose that non-faulty P_i diffuses a double-signed m at real-time T to another non-faulty P_k which receives m at $T+\delta_m$, where $0 \leq \delta_m < \delta$. For $d \geq \delta/(1-5\rho)$, $START_k(P_j{:}P_i, \geq m.TS) - T \geq \delta_m$ is true in any execution of the early-order protocol.

Proof: Let us abbreviate $START_k(P_j{:}P_i, \geq m.TS)$ as simply $START_k(P_j{:}P_i)$. Let m', $m'.TS \geq m.TS$, be the message whose acceptance by P_k scheduled the execution of the *update* process that sets $PC_k[P_j{:}P_i] \geq m.TS$ at $START_k(P_j{:}P_i)$. As in lemma S2, we will prove this lemma by considering three cases of the *originator* of m' being (i) P_k, (ii) P_i, and (iii) P_j. Recall that from $START_i(P_j, \geq m.TS)$ onwards, P_i will not accept P_j's single-signed messages with timestamp smaller than or equal to $m.TS$. So, $T \leq START_i(P_j, \geq m.TS)$; also $\delta_m < \delta$. This means that this lemma can follow from lemma S2 for those cases whose correctness arguments do not involve the bounds of table 1 that get changed in table 3. It can be seen that the timeliness bounds of table 1 that get referred to in the correctness arguments for cases (i) and (ii) remain the same in table 3 as well. So, we will omit the first two cases and prove the lemma only for case (iii):

(iii) the *originator* of $m' = P_j$. Note that the *originator* of m is also P_j. The lemma is trivially true if m and m' are identical; let us suppose that they are not. Let T' be the real-time when P_k received m'; obviously $T' \leq START_k(P_j{:}P_i)$. If $path(m') \equiv path(m)$, then, by rule 5, P_i must have diffused to P_k first m and then m', and $m'.TS > m.TS$. Since communication between non-faulty processors is in fifo order, $T' > T + \delta_m$. So, lemma is true for $path(m') \equiv path(m)$.

Let us finally consider the possibility of $path(m') \neq path(m)$; i.e., m' is single-signed. As per the entry C3 of Table 3 with suffixes k and i interchanged, $T' = START_k(P_j{:}P_i)\text{-}3d(1+\rho_k)$. P_i receives the diffused m' from P_k before $T'+\delta$. If $T < T' + \delta$ then $START_k(P_j{:}P_i) - T > 3d(1+\rho_k) - \delta$; so, $START_k(P_j{:}P_i) - T > \delta$ if $d \geq \delta/(1\text{-}\rho)$. Let us suppose that $T \geq T' + \delta$; i.e., P_i receives double-signed m' (from P_k) before it accepts m from P_j and diffuses it to P_k. Given that P_i accepts m that is received after m', $m'.TS \geq m.TS$, and the *immediate_sender* of the double-signed m' is non-faulty, the premises of lemma 5 are met. So, P_i must find m' timely. Since m is also found timely, the entry D2 of table 3 indicates that m must have been received by P_i within $d(1+\rho_i)$ real-time after m' is received. So, $T < T' + \delta + d(1+\rho_i)$. Substituting for T' in $T' = START_k(P_j{:}P_i)\text{-}3d(1+\rho_k)$: $T < START_k(P_j{:}P_i) - 3d(1+\rho_k) + \delta + d(1+\rho_i)$; $START_k(P_j{:}P_i)\text{-}T > \delta$ whenever $d \geq \delta/(1+3\rho_k/2\text{-}\rho_i/2)$, which is true since $d \geq \delta/(1\text{-}5\rho)$. Hence the lemma. q