

School of Computing Science,
University of Newcastle upon Tyne



A Timeout-Based Message Ordering Protocol for a Lightweight Software Implementation of TMR Systems

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

Technical Report Series

CS-TR-817

October 2003

Copyright©2003 University of Newcastle upon Tyne
Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

A Timeout-Based Message Ordering Protocol for a Lightweight Software Implementation of TMR Systems

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

[†]School of Computing Science, University of Newcastle upon Tyne, UK

[‡]Universidade Federal de Campina Grande, Paraíba, Brazil

Abstract

Replicated processing with majority voting is a well known method for achieving reliability and availability. 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 input requests 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 does not require clocks to be kept in bounded synchronism. Our design efforts focus on keeping the ordering delays small, without unnecessary increase in message overhead. Consequently, we are able to show that no symmetric protocol that works only with unsynchronised clocks, can provide a smaller worst-case delay. We also demonstrate, through analysis and experiments, that our protocol is faster than a time-based one of identical message complexity in certain situations which can prevail in many application settings.

Keywords and Phrases

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

1. Introduction

We consider the task of designing and implementing a system that continues to provide services in the presence of a bounded number of faulty processors. The weaker the assumptions made about a faulty processor's behaviour, the wider is the set of faults tolerated. The weakest fault model known to the fault-tolerant community is the *Byzantine* model [1], and N modular redundant (NMR) processing is one of the most effective ways to mask the effects of Byzantine failures [2]. The basic idea here is to use N , $N \geq 3$, processors in place of a single processor so that failures of at most $(N - 1)/2$ processors are masked. A

Triple modular redundant (TMR) system with $N = 3$ is the most practical version of an NMR system. The three processors of a TMR system execute every given task in parallel and the results produced by them are subject to a majority vote. If the voted result is regarded as the output from the TMR system, the system functions correctly provided (i) at least two of its constituent processors are non-faulty, (ii) all non-faulty ones produce identical results after executing any given task, and (iii) the voting performed is correct.

Process replication within a TMR system assumes the well understood state machine model, which imposes two requirements [3] on replicas: they must be deterministic (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result) and they must start in the same state. The second requirement means that a message ordering mechanism that presents the input messages to replicas in an identical order is necessary. In this paper, we develop an ordering protocol for the *authenticated Byzantine* fault model – perhaps the most well-known sub-class of the Byzantine fault model – in which a faulty processor cannot undetectably forge a non-faulty processor’s signature; the protocol also assumes the following *synchronous* environment: (i) communication delays between non-faulty processors are bounded by a known constant; (ii) processing and scheduling delays within a non-faulty processor can also be bounded; and, (iii) each non-faulty processor has a local read-only physical clock whose running rate with respect to the passage of real time differs from unity by a small and known bound. (Real time is measured in an assumed Newtonian time frame that cannot be directly observed.)

Message ordering in the presence of failures requires the ability to detect late and absent messages. In the time-based approach [4], this capability is usually achieved by synchronising non-faulty processors’ clocks within some known constant. Since non-faulty processors need not have clocks with identical running rates, they should periodically execute a synchronisation protocol (e.g., [5, 6]) to adjust their clock readings by appropriate amounts. This synchronisation involves (i) periodic exchange of messages, consuming network bandwidth, (ii) using data abstractions [7] to adjust the readings of a physical clock (a read-only object), and (iii) implementing amortisation techniques [8] to avoid sudden jumps in the synchronised clock readings. An alternative to building a heavy-weight, synchronised clock abstraction is to use timeouts and to employ (unsynchronised) physical clocks only for measuring timeouts. We adopt here the timeout-based approach for designing the TMR message ordering protocol. Specifically, we derive timeout durations called the *timeliness bounds* to detect late and absent messages; such bounds are derived by taking advantage of

the synchronous nature of the TMR system and they remain constant while the system is operational.

The works of [9] and AMp [10] suggest that using timeouts in the synchronous context is not something new. One of the designers of AMp analysed time vs. timeout-based approaches and observed [11]: though the synchronous, timeout-based protocols cannot be perfect substitutes for their time-based counterparts in *all* circumstances, they can however provide attractive alternatives in a number of application settings. We analytically identify, and experimentally observe, the situations where our timeout-based protocol is faster than its time-based counterpart. Existence of such situations indicates that the protocol presented here enhances the tool-kit available to a TMR system builder.

The rest of the paper is organised as follows. Section 2 describes the structure of the TMR system, the basic assumptions and the message ordering requirements. Section 3 develops and presents the protocol. Section 4 outlines the design efforts involved in keeping the ordering delays small. Section 5 presents proofs of correctness and also proves that no (symmetric) protocol that works only with unsynchronised clocks can guarantee smaller worst-case ordering delays. In section 6, we compare the ordering delays in both time and timeout based approaches, and identify situations where the timeout protocol performs faster than the time-based protocol; our implementation experiments demonstrate that such situations are not uncommon in practice. Section 7 concludes the paper after a brief survey on related work.

2. System Description and Assumptions

The TMR system 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 other two processors of the system by *internal* links. Also, it is connected to the ‘outside world’ or the system environment from which input messages are received and output messages are sent to. Figure 1 shows a TMR system whose processors are connected to the environment via a bus. The unit (shown as a black square in the figure) that connects a processor to the bus is called the network attachment controller (NAC).

Assumption 1: Within a TMR system, processors fail independent of each other and at least 2 processors are non-faulty (and therefore do not fail).

Assumption 2: The internal links that connect processors do not fail. Within a processor, the messages received via each internal link are queued separately before they are processed.

A processor is reliably connected to the system environment via a NAC which does not allow the attached processor to use the bus continuously for a long time. Thus, if a faulty processor, like a “babbling idiot”, transmits randomly generated messages, it cannot overwhelm the communication subsystem and prevent a non-faulty processor from receiving another non-faulty processor’s messages nor from accessing the bus.

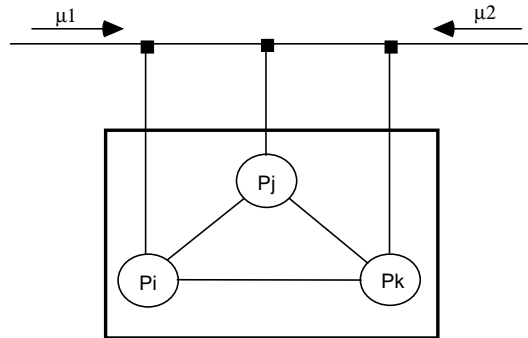


Figure 1. The TMR system.

Assumption 3: Each processor can sign the messages it sends, and authenticate the signed messages it receives [12] such that:

- (a) A non-faulty processor’s signature for a given message is unique and cannot be generated by any other processor.
- (b) Any attempt to alter the contents of a non-faulty processor’s signed message is detected by any other non-faulty processor.

These assumptions define the authenticated Byzantine fault model. In the general (unauthenticated) Byzantine model, the assumed failure modes are described not by listing how a faulty processor is expected to fail but rather by describing a small set of incorrect behaviour that are disallowed: a faulty processor cannot (i) make a non-faulty one faulty (failure independence), and (ii) overwhelm the communication subsystem with random messages and deny a non-faulty processor the capability to communicate with another non-faulty processor or with the environment. The authenticated model additionally restricts a faulty processor’s ability to undetectably impersonate a non-faulty processor (assumption 3). Byzantine fault models are regarded to be the weakest since they allow a faulty processor to be operated by a malicious adversary who can exercise any combination of permitted failure modes to make non-faulty processors order inputs differently.

The next three assumptions define the synchronous nature of the system. Before stating them, we remark that we write, throughout this paper, real time values in Greek letters and

clock time values in italicised lower case Roman letters; the term ‘clock’, when not qualified, will refer to a processor’s hardware clock.

Assumption 4: If any non-faulty processor prepares and transmits m at real time τ , any non-faulty destination will receive m at real time τ' , $\tau \leq \tau' < \tau + \delta$, where δ , $\delta > 0$, is a known constant. Thus, δ bounds the message queuing delays at the sending and receiving processors, and the propagation delay from the sender to a receiver.

Assumption 5: The clock of any non-faulty processor, say P_x , measures a duration of length l in real time $l(1 + \rho_x)$, where $|\rho_x| \leq \rho$ and ρ is a known positive constant. (ρ is around 10^{-6} for non-faulty processor clocks.)

Assumption 6: Processing and scheduling delays are bounded and known for any non-faulty processor; more precisely, any non-faulty processor: i) performs a local computation (e.g., processing a received message) within a known amount of time; and, ii) schedules a computational task within some known amount of time.

Remarks: Assuming the synchronous model requires accurate estimation of certain bounds that must hold throughout the system operation. Since these bounds are assumed to hold for all non-faulty processors, a violation of any of these bounds makes a non-faulty processor appear to be behaving like a faulty one; e.g., a violation of assumption 4 means that the (non-faulty) sender appears to be faulty to the (non-faulty) receiver. When the fault hypothesis (of at most one fault) is thus artificially undermined, processors that correctly execute the protocol can end up ordering messages differently. Therefore, adopting the synchronous model, be it for time-based or timeout-based approach, requires defining the maximum processing and communication load which the system will ever be subject to. This could in turn involve assessing the maximum processing load which a valid input can impose on processors, and the number of most-demanding inputs which the system has to concurrently deal with at any given time. The only alternative to having to determine hard bounds on various delays is to opt for the *asynchronous* model which is briefly described in Section 7.

Note that when the actual system load is less than the maximum defined, the processing, the scheduling and communication delays will be smaller than their respective upper bound. A question then naturally arises: whether an order protocol can order messages faster when the TMR system is lightly loaded. A time-based protocol, by the nature of its design, cannot work faster, while a timeout-based can. That is why our protocol works faster than its time-

based counterpart when certain situations prevail, one of which happens to be a lightly loaded TMR. However, for this benefit to manifest, the design efforts, as shown in section 4.3, must go farther than those needed for a time-based protocol. (We remark here that the class of early-stopping protocols, such as [13, 14], are designed to terminate early not when delays are smaller than expected but when fewer failures than expected occur.)

2.1 Input Message Ordering

Non-faulty processors can receive input messages in different order. Referring to figure 1, P_1 can receive μ_1 followed by μ_2 and P_k in the reverse order. So, when a processor receives an input message from the system environment, it must first decide the processing order for that message. For that, it forms an *internal* message m that contains the received input message in the data field $m.\mu$ and sends m to all other processors in the system using the order protocol that guarantees the following two conditions:

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

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

These conditions ensure that an input μ supplied to a non-faulty P gets identically ordered in the form of P 's internal message m , $m.\mu = \mu$, by all non-faulty processors of the TMR system within Δ . We refer the reader to [15] for details on how a non-faulty processor (i) derives an ordered stream of inputs from an ordered stream of internal messages, and (ii) generates a voted output from the results it computes. This paper will focus only on the (timeout-based) ordering of internal messages. Note that since the TMR system can have at most one faulty processor, an input μ must be supplied to at least two processors within the system; we will assume that every μ is sent to all three processors in the TMR system.

3. The protocol

The protocol has three aspects to it: (i) *message counters* maintained by processors, (ii) *message diffusion* that enables non-faulty processors to receive each other's messages, and (iii) *timeliness checks* to assess the timeliness of a received message. All processors execute the same version except for processor identities and signatures. The protocol is described for P_1 which is assumed to be non-faulty throughout the paper.

3.1. Message Counter and Diffusion

P_i maintains a counter called the *message counter*, denoted as MC_i , which holds an integer value and is initialised to $INIT_VAL$ (usually 1) when the system is first started. For every input μ received from the environment, P_i forms an internal m in the following manner: the data field $m.\mu$ is set to μ , the originator field $m.O$ to i ; the timestamp field $m.TS$ to MC_i ; further, MC_i is incremented by 1. Incrementing MC_i immediately after timestamping m ensures that any message P_i later forms gets a timestamp larger than $m.TS$. An internal message m formed is accepted with its copy being entered into a message list called the *accepted_i* and is then sent to other processors using the *send(m)* primitive. An invocation of this primitive discards m if $m.S$ contains two signatures, and performs the following actions if $m.S$ contains no or one signature: signature of P_i for m is generated and appended to any signature that may already be in $m.S$; the signed m is then transmitted to processors in the system that have not signed m . P_i receives internal messages from other processors by executing the *receive(m)* primitive which blocks until it can return an authentic m that is received via an internal link and has the authentic signatures of one or two distinct processors other than P_i .

Whenever P_i receives m , it checks whether m is timely. (Procedures for checking the timeliness of a received message are described in the next sub-section.) If m is found untimely, it is discarded; otherwise, the following three actions are performed: (i) MC_i is set to the *maximum of* $\{MC_i, m.TS + 1\}$ ¹, (ii) m is *accepted* by entering a copy of m into *accepted_i*; and, (iii) *send(m)* which ensures that if the accepted m had one signature when it was received, it is diffused to the processor that appears not to have ‘seen’ m . Note that the message diffusion in (iii) is essential to ensure the unanimity condition when one processor can be faulty [16].

From the description above, it is obvious that: (i) a message sent or received by a non-faulty processor will have either one or two authentic signatures; and (ii) every sent message, but no received message, carries the host processor’s signature. For a given message m , *path(m)* is defined as the ordered sequence of processors that 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*

¹ A faulty processor can give arbitrarily large $m.TS$ to messages it forms and sends, and thereby cause MC_i to wrap around frequently. The water-mark scheme of [17] can be used to restrict this incorrect behaviour.

sender of m . Note that the *originator* and the *immediate sender* of m are one and the same if m is single-signed. Two paths are said to *intersect* if they contain one or more processors in common.

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, and identical for all non-faulty processors of the system. 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 .)

Suppose that P_i receives a message m at its local clock time t_i . There can arise one of three possible situations depending on the value of MC_i at t_i : $MC_i < m.TS$ or $MC_i = m.TS$ or $MC_i > m.TS$. If $MC_i < m.TS$ or $MC_i = m.TS$ then m is a ‘future’ or a ‘present’ message respectively and is 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 should be judged based on how much time has elapsed since MC_i first became larger than $m.TS$. So, timeliness checks are needed only for messages received with past timestamps. Let us suppose that the m received at t_i is a past message. Let 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 P_i accepted m' , $MC_i \leq m.TS$. So, $m'.TS \geq m.TS$ must be true. Note that m' could have been formed by any processor including P_i , therefore $path(m')$ can be any one of $P_i, P_j, P_k, P_j:P_k$ or $P_k:P_j$. Let P_i accept m' at its clock time t_i' , $t_i' < t_i$. For m to be considered timely, $(t_i - t_i')$ must be less than a fixed bound, called the *timeliness bound*, whose value depends on $path(m')$ and $path(m)$. Table 1 summarises these timeliness bounds for all combinations of $path(m')$ and $path(m)$. For example, if m' is a single signed message from P_k and m is a single signed message from P_j , then the entry B2 (row B, column 2) indicates that $(t_i - t_i')$ must be less than $2d$ for P_i to consider m timely. We will denote an entry of Table 1 by treating the table as a matrix: $Table1[p_r, p_c]$ denotes the value in the row corresponding to the path p_r and in the column corresponding to the path p_c .

$path(m)$ $path(m')$	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 a past m given that MC_i first exceeded $m.TS$ due to accepting m' .

3.3. Protocol Description

To perform the timeliness checks on received messages, P_i maintains path counters, denoted as $PC_i[p]$, for every path p through which P_i can receive a message. These counters are initialised to $(INIT_VAL - 1)$ and are updated at appropriate times using two primitives: an invocation of $update(path: p, timestamp: T)$ sets $PC_i[p] = \text{maximum of } \{PC_i[p], T\}$; $schedule\ update(p, T)$ at t schedules $update(p, T)$ to be invoked at (local) clock time t , if t is a future time when the $schedule$ instruction is executed.

Whenever P_i forms and sends m' or receives a timely m' , updates are scheduled to set $PC_i[p] = \text{maximum of } \{PC_i[p], m'.TS\}$, for every $p \in \{P_j, P_k, P_j:P_k, P_k:P_j\}$, after the elapse of time intervals indicated in $Table1[path(m'), p]$. Thus, the timeliness of a received m can be checked simply by referring to $PC_i[path(m)]$: m is timely only if $m.TS > PC_i[path(m)]$ when m was received. Putting it differently, a received m is not timely if $PC_i[path(m)] \geq m.TS$ when m was received. Let $PC_{i,min}$ denote the minimum of the path counter values at any time, and its current value be T . From the moment $PC_{i,min}$ becomes T , the set of messages in $accepted_i$ which have the timestamp of T cannot expand in its size; that is, this set has *stabilised* and is denoted as $stable_i(T)$.

To perform message ordering, entries of $stable_i(T)$ are first stripped of their signatures, and then duplicate entries are discarded. 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 are generated when the faulty processor gives the same timestamp to distinct messages m and m' it forms. Spurious messages in $stable_i(T)$ are discarded. After this pruning, $stable_i(T)$ contains at most one message originating from a given processor. Messages of $stable_i(T)$ are ordered (i) according

to their source processors, and (ii) only after the contents of all $stable_i(T^*)$, $T^* < T$, have been ordered. To ensure (ii), a *stability counter*, denoted as SC_i and initialised to $(INIT_VAL - 1)$, is used. The protocol (for P_i) is presented in figure 2.

4. Estimating Timeliness Bounds

Central to protocol design is the estimation of timeliness bounds which need to be (i) as small as possible for message ordering to be fast, and (ii) large enough to ensure that the validity and unanimity conditions can be realised. This section describes how we meet these competing requirements. It is done in three parts.

```

do /* Broadcast process: handles external messages */
  get  $\mu$  from outside the system;
   $m.\mu = \mu$ ;  $m.TS = MC_i$ ;  $m.O = i$ ;
   $MC_i = MC_i + 1$ ;
   $t = clock_i$ ;
  for every  $p \in \{P_j, P_k, P_j:P_k, P_k:P_j\}$  do
    schedule update( $PC_i[p]$ ,  $m.TS$ ) at  $t + Table1[P_i, p]$ ;
  end for
  send( $m$ );
   $accepted_i = accepted_i \cup \{m\}$ ;
end do /* Broadcast process */
||
do /* Diffuse process: handles internal messages */
  receive( $m$ );
  if ( $m.TS \leq PC_i[path(m)]$ ) then discard( $m$ )
  else
     $MC_i = \text{maximum of } \{MC_i, m.TS + 1\}$ ;
     $t = clock_i$ ;
    for every  $p \in \{P_j, P_k, P_j:P_k, P_k:P_j\}$  do
      schedule update( $PC_i[p]$ ,  $m.TS$ ) at  $t + Table1[path(m), p]$ ;
    end for
    send( $m$ );
     $accepted_i = accepted_i \cup \{m\}$ ;
  end if
end do /* Diffuse process */
||
do /* Order process: identifies and orders stable messages */
   $PC_{i,min} = \text{minimum of } \{PC_i[p], \forall p \mid p \in \{P_j, P_k, P_j:P_k, P_k:P_j\}\}$ ;
  while  $SC_i \leq PC_{i,min}$  do
     $SC_i = SC_i + 1$ ;
     $stable_i = \{m \mid m \in accepted_i \text{ and } m.TS = SC_i\}$ ;
     $accepted_i = accepted_i - stable_i$ ; /* remove stable messages */
    if  $stable_i \neq \emptyset$  then
       $distinct = \emptyset$ ;

```

```

for every  $m$  in  $stable_i$  do
   $m.S = \emptyset$ ; /* strip off signatures from  $m$  */
  if  $m \notin distinct$  then insert  $m$  in  $distinct$  end if
end for
 $stable_i = distinct$ ;
 $spurious = \{m, m' \mid m \in stable_i \wedge m' \in stable_i \wedge m'.O = m.O \wedge m'.\mu \neq m.\mu\}$ ;
 $stable_i = stable_i - spurious$ ; /* remove spurious messages */
if  $stable_i \neq \emptyset$  then  $\forall m \mid m \in stable_i$  deliver  $m$  ordered by  $m.O$  end if
end if
end while
end do /* Order process */

```

Figure 2. The Timeout-based Protocol for Message Ordering.

First, the basic condition that should be met for the protocol to be correct is stated. We then analyse how meeting this condition is made difficult by various possible failure modes of the faulty processor; this analysis leads to the identification of a requirement for meeting the basic condition.

The third part is on estimating timeliness bounds which meet that requirement and is done in two stages. First, we derive the bounds in a manner that is typically employed in designing time-based protocols. This results in what we call the ‘conservative’ bounds. Next, these bounds are reduced, which leads to smaller worst-case ordering delay.

4.1. The Basic Condition

The following condition must be met for identical ordering by non-faulty processors:

Unanimous acceptance: m enters the *accepted* list of a non-faulty processor if and only if m or a message *equivalent* to m enters the *accepted* list of every other non-faulty processor.

A message equivalent to m is denoted as $equiv(m)$ and is defined as a message that differs from m only in its signature field; that is, $equiv(m).\mu = m.\mu$, $equiv(m).O = m.O$, $equiv(m).TS = m.TS$ and $equiv(m).S \neq m.S$. For every accepted m , (non-faulty) P_i can receive at most one $equiv(m)$. Note that $equiv(m)$ and m become identical once their signatures are stripped off (i.e., once the signature fields are set to \emptyset).

If the above condition holds, then non-faulty processors will construct an identical $stable(T)$ of signature-stripped and non-spurious messages for every given T , $T \geq INIT_VAL$. Given that processor ordering is unique and known to non-faulty processors, the order determined by them for any (signature-stripped) m will be identical.

4.2. Effects of Failures

A faulty processor, say P_j , can attempt to prevent the above condition from being met, by failing in the following ways.

F1 (*impersonating a non-faulty processor*): P_j generates a signed message on behalf of P_i and attempts to deceive P_k into accepting the forged message.

F2 (*delayed sending of own messages*): P_j delays the sending of a message m it generates, with the consequence that one non-faulty processor, say P_i , finds m timely and P_k does not. This situation, depicted in figure 3(a), has the effect of m entering $accepted_i$ but not $accepted_k$.

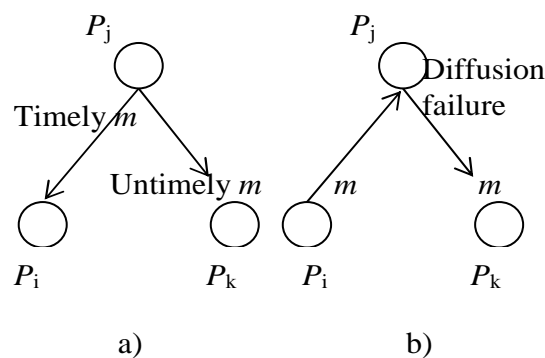


Figure 3. P_j 's failures of types F2 and F4.

F3 (*two-facing while sending own messages*): P_j sends a properly signed m to, say, P_i , and sends to P_k either (a) an inauthentic version of m , or (b) nothing, or (c) a different, authentic message m' (that is never sent to P_i). The effect of a two-facing failure is the same as the previous category: a 1-signed message enters the *accepted* list of only one non-faulty processor.

F4 (*failure during diffusion*): P_j fails while diffusing a message m which it received from, say, P_i as shown in figure 3(b). It can fail by altering the contents of m (call this failure type F4.1) or by delaying the diffusion by an arbitrary amount of time (call this type F4.2). Assumption 3 reduces the impact of F1 and F4.1 into P_k detecting and discarding P_j 's tampered message as not authentic. F4.2 can result in P_k not receiving the diffused message at all, or receiving it but finding it untimely. Thus, P_j 's failures of types F2, F3, and F4.2 can cause P_j 's m not to be accepted by one non-faulty processor while m or $equiv(m)$ is being

accepted by another. Despite this, the condition of unanimous acceptance needs to be satisfied through message diffusion which is feasible if the following requirement is met:

Unanimous acceptance requirement: a non-faulty processor finds a received m timely if the immediate sender of m is non-faulty.

Suppose that the above requirement is met. In case of F2 and F3 type failures, if the 1-signed m sent by faulty P_j enters $accepted_i$ but not $accepted_k$, then the $equiv(m)$ diffused by P_i will be accepted by P_k ; P_j 's failures of type F4.2 are made irrelevant since m sent by non-faulty P_i is assured to enter $accepted_k$.

4.3. Derivation of Timeliness Bounds

As per the protocol, whenever P_i accepts m , MC_i is immediately set to *maximum of* $\{MC_i, m.TS+1\}$, while $PC_i[p]$, for every path $p \in \{P_k, P_j, P_j:P_k, P_k:P_j\}$, is set to *maximum of* $\{PC_i[p], m.TS\}$ after some time. So, $MC_i > PC_i[p]$ is always true for any path p . This means that when a received m is a future or present message (i.e., $MC_i \leq m.TS$), it is found timely (i.e., $PC_i[p] < m.TS$) and therefore the unanimous acceptance requirement is trivially met. So, in what follows, we show that the timeliness bounds of Table 1 meet the above requirement when a received m is a past one.

We will present our derivations in the context in which the bounds of Table 1 are presented: P_i is non-faulty and accepts m' at its clock time t_i' and receives m , $m.TS \leq m'.TS$, at t_i , $t_i > t_i'$; just before t_i' , $MC_i \leq m.TS$, and at t_i' $MC_i > m'.TS$. To keep the derivation simple, we assume the following. (i) The minimum message transmission delay (as measured by a non-faulty clock) is zero. (ii) The clocks of all non-faulty processors have an identical running rate; so, $d = \delta/(1-\rho_i)$. (This assumption is removed in subsection 4.3.3.) (iii) Unless stated explicitly, time is measured according to the clock of 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. (iv) Finally, the subscript i is dropped from t_i and t_i' , where the context is obvious.

4.3.1. Conservative bounds

Lemma 4.1. Given that P_i accepts m' at t' , any non-faulty P_x will have $MC_x > m'.TS$ before $t' + d$.

Proof: If P_x has signed m' , then $MC_x > m'.TS$ when it signed m' . Since P_x 's signing of m' has to be prior to P_i accepting m' , the lemma is true. If P_x has not signed m' , it will

receive the diffused $equiv(m')$ from P_i before $t' + d$. Upon the reception of m' , if $MC_X \leq equiv(m').TS$ then $equiv(m')$ will be seen by P_X as a present or future message, and therefore must be accepted which will then set $MC_X > m'.TS$. \square

Say, non-faulty P_k forms and sends m , $m.TS \leq m'.TS$. By Lemma 4.1, MC_k becomes larger than $m'.TS$ before $t' + d$. So, m should have been sent by P_k before $t' + d$. Even if m experiences the maximum delay of just less than d time, P_i should receive P_k 's single-signed m before $t' + 2d$. It suggests that if P_i receives a single-signed message m at t such that $(t - t') < 2d$, it must consider m timely. This scenario is depicted in figure 4(a) where time progresses from left to right and the labels " $>m'.TS$ " and " $=m.TS+1$ " indicate the earliest instances when a non-faulty message counter exceeds $m'.TS$ and when it becomes equal to $m.TS + 1$, respectively; " $<nd$ " labels an interval of length less than nd , for some integer $n \geq 1$. On the time-line of P_k (in figure 4(a)), the timing instance labelled " $=m.TS+1$ " cannot be on the right hand side of that labelled " $>m'.TS$ " since $m.TS \leq m'.TS$. So, P_i can receive a timely m from P_k at t , $(t - t') < 2d$.

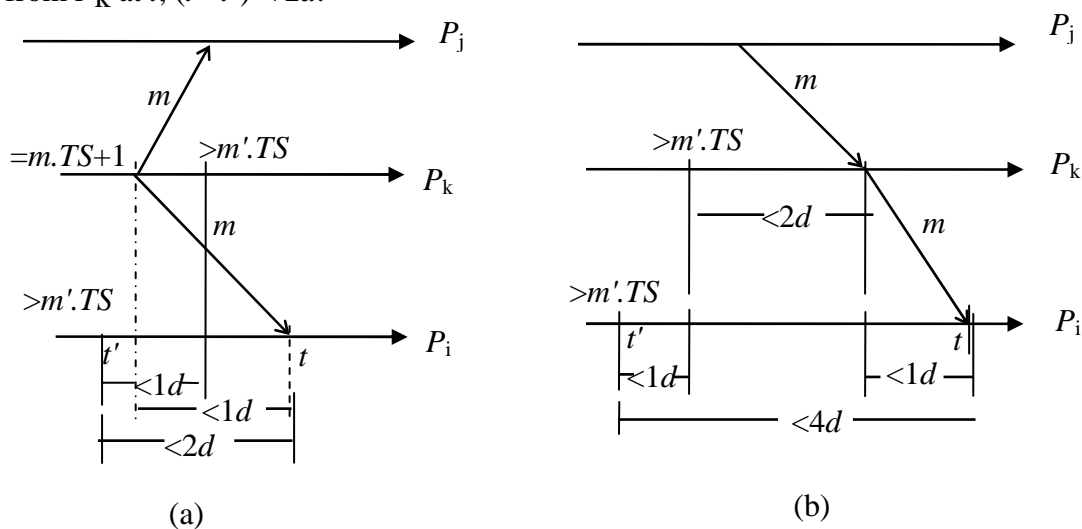


Figure 4. (a) Non-faulty P_k forms and sends m . (b) P_k diffuses P_j 's m .

Suppose that m originates from a faulty P_j and that P_k receives it within $2d$ time after $MC_k > m'.TS$ has become true. (See figure 4(b).)

Just like in figure 4(a) where P_i accepts P_k 's m because m arrives within $2d$ time after $MC_i > m'.TS$, P_k must now consider P_j 's m timely. (Note that a non-faulty processor cannot know whether another processor is non-faulty or faulty.) When P_k diffuses m , P_i must find the diffused message timely for the unanimous acceptance requirement to be met. From figure

4(b), P_i can receive the diffused m at t , $t < t' + 4d$. So, P_i 's timeliness check for accepting a double signed m is $(t - t') < 4d$.

4.3.2. Reducing the conservative bounds: why and how?

Let us evaluate Δ had the conservative bounds been used as such. Consider the following scenario: P_i receives and accepts 1-signed m' from non-faulty P_k at time t' . P_i would have to wait until $t' + 2d$ and $t' + 4d$ to deduce that it would no longer accept a 1-signed m and a 2-signed m , $m.TS \leq m'.TS$, respectively. That is, P_i could not stabilise m' no earlier than $t' + 4d$. Say, P_k sends m' at t'_k , $t'_k \leq t' \leq t'_k + \delta$. So, the delay (measured from t'_k) for P_i to order m' could be at most $\delta + 4d$. The maximum delay for P_k to order its own m' is $4d$. Thus, Δ , the maximum ordering delay, would have to be $\delta + 4d$. Interestingly, the case which decides Δ is P_i ordering m' , $m'.O \neq P_i$; it turns out that if $m'.O \neq P_i$, the conservative timeliness bounds can be reduced, permitting P_i to stabilise m' at $t' + 3d$ itself, resulting in a smaller $\Delta = 4d$. Below, we present the intuition behind this reduction.

Suppose that P_i has received and accepted m' at t' , and that $P_i \neq m'.O = P_k$ (say). P_i can now expect any of P_k 's 1-signed m , $m.TS < m'.TS$, to be received before $t' + d$, if P_k is non-faulty. (Note: the unanimous acceptance requirement needs to be met only when P_k , the immediate sender of m , is non-faulty.) This is because, non-faulty P_k sends the messages it forms, in the increasing order of message timestamps; P_i must receive the earlier one within at most d after it has received the later one. So, the conclusion is: given that m' is accepted at t' , the timeliness bound for a 1-signed m such that $m.TS \leq m'.TS$ and the *originator* of $m' \equiv \text{originator of } m \neq P_i$, is only d (see entries B1 and E1 of Table 1), and need not be $2d$ which is the conservative bound for any 1-signed m .

Observe that we achieved the above reduction (for two particular cases) by considering the slowest possible behaviour of an unknown m whose immediate sender is non-faulty and $m.TS \leq m'.TS$; further, we also made use of two facts: (i) $m'.O \neq P_i$, and (ii) $\text{path}(m')$ intersects with $\text{path}(m)$. It turns out that whenever (i) and (ii) hold, reduction is possible. Intuitively, the latest arrival time of the unknown m can be estimated more precisely, if m and the accepted m' are handled by the same processor. Because there are only three processors in the TMR system, $\text{path}(m')$ and $\text{path}(m)$ always intersect if m is 2-signed and $m'.O \neq P_i$. Furthermore, if $\text{path}(m')$ and $\text{path}(m)$ have only one common processor, then that processor

does not have to be non-faulty for reduction to be feasible. Suppose that P_j is faulty. The only path combination for which $path(m')$ and $path(m)$ intersect only at P_j is: $path(m') = P_j$ and $path(m) = P_j; P_k$, given that the immediate sender of m (i.e., P_k) has to be a non-faulty one. Appendix A argues that the bound for this path combination is $3d$ as indicated by the entry C3 of Table 1. With reduced bounds in use, P_i , after having accepted m' at t' , can conclude by $t' + 3d$ that it would no longer accept any m whose immediate sender is non-faulty and $m.TS \leq m'.TS$, i.e., it can stabilise m' no later than $t' + 3d$.

Using arguments similar to those used in Appendix A, we have reduced many other bounds, all of which are indicated by those entries of Table 1 that are neither $2d$ nor $4d$, and by entries D3 and E4 which have $2d$ for a double-signed m . These arguments are shown in [18].

4.3.3. Accounting for non-identical clock rates

We now remove the following simplifying assumption made earlier: clocks of all non-faulty processors have a known, identical running rate. Recall that d was defined in Section 3.2 to be the clock time interval such that by measuring d in its local clock a non-faulty processor is guaranteed to have measured a real time interval of at least δ . Since the running rate of any non-faulty clock can differ from unity by at most ρ (assumption 5), it is enough to set $d = \delta/(1 - \rho) = d_1$ (say), if all non-faulty clocks had the same unknown running rate. We argue below that the effects of non-identical running rates of non-faulty clocks are compensated for, when d_1 is increased to $d_1(1+2\rho)(1+2\rho)$.

Suppose that all processors start measuring d_1 from the same real-time instance by referring to their respective local clocks. If non-faulty clocks had been running at an identical rate, then every given non-faulty processor deduces the following two facts when it completes measuring d_1 . *Fact-1*: every other non-faulty processor has completed measuring d_1 , and *fact-2*: every non-faulty processor deduces that every other non-faulty processor has completed measuring d_1 . By assumption 5, non-faulty clocks can drift apart at the maximum rate of 2ρ . So, a non-faulty processor must measure $d_2 = d_1(1+2\rho)$ in its clock to ascertain that every other non-faulty processor must have measured at least d_1 . That is, every non-faulty processor must measure d_2 in its clock to be able to deduce *fact-1*. Similarly, to be able to deduce *fact-2*, a duration $d_3 = d_2(1+2\rho)$ must be measured in the local clock. More precisely, only after measuring d_3 in its local clock, a given non-faulty processor, say P_j , can (i) ascertain that

every other non-faulty processor, say P_k , must have measured at least d_2 and (ii) deduce that P_k would have deduced *fact-1*.

If we ignore the terms containing the second and higher order powers of ρ , we can write $d_3 = d_1/(1-4\rho) = \delta/(1 - 5\rho)$. Thus, choosing $d \geq d_3 = \delta/(1 - 5\rho)$ will account for non-identical running rates of non-faulty clocks.

5. Protocol Correctness and Optimality Result

Lemma 5.1: The protocol guarantees the validity condition with $\Delta = 4d(1 + \rho)$, provided $d \geq \delta/(1 - 5\rho)$.

Proof: Consider an execution of the protocol in which P_i and P_k are non-faulty. Let P_i form and send a message m at real-time τ_i . P_k receives m before $\tau_i + \delta$ (by assumption 4). The arguments of Section 4.3 indicate that the timeliness bounds of Table 1 satisfy the unanimous acceptance requirement when $d \geq \delta/(1 - 5\rho)$. (For a formal proof of correctness, see Appendix C.) So, P_k will find the received m timely and accept it. Note that when a non-faulty processor accepts a message m , none of its path counters stays below $m.TS$ after some finite time. So, the message m accepted by P_i and P_k , will be taken up for ordering and 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 within a finite time after τ_i .

The *schedule* instructions in the *Broadcast* process indicate that $PC_{i,\min}$ reaches $m.TS$ within $4d$ clock time after τ_i . So, P_i orders m no later than $\tau_i + 4d(1 + \rho_i)$. The *schedule* instructions in the *Diffuse* process indicate that $PC_{k,\min}$ reaches $m.TS$ within $3d$ clock time after P_k receives m which is before $\tau_i + \delta$. So, P_k orders m no later than $\tau_i + \delta + 3d(1 + \rho_k)$. Thus the message m sent by P_i gets ordered by P_i and P_k , no later than $\tau_i + \text{maximum of } \{4d(1 + \rho_i), \delta + 3d(1 + \rho_k)\}$. By assumption 5, $|\rho_i| \leq \rho$ and $|\rho_k| \leq \rho$. So, $\Delta = \text{maximum of } \{4d(1 + \rho), \delta + 3d(1 + \rho)\} = 4d(1 + \rho)$. Hence the lemma. \square

Let $Stable_i(T)$, for some T , $T \geq INIT_VAL$, denote the set *stable_i* which non-faulty P_i first constructs during an execution of the protocol when $SC_i = T$. The code for the Order process indicates that (a1) P_i strips off signatures of every m in $Stable_i(T)$ and removes duplicates;

and then, (a2) it removes spurious messages from $Stable_i(T)$. Let $SigFree_i(T)$ and $SpuFree_i(T)$ denote the resulting $Stable_i(T)$ after a1 and a2 are carried out, respectively:

Definition 5.1: $SigFree_i(T) = \{m \mid m \in Stable_i(T) \wedge m.S = \emptyset\}$.

Definition 5.2: $SpuFree_i(T) = \{m \mid m \in SigFree_i(T) \wedge (\neg \exists m' \in SigFree_i(T): m.O = m'.O)\}$.

Lemma 5.2: Consider an execution of the protocol in which P_i and P_k are non-faulty. Suppose that: $m \in Stable_i(T) \Rightarrow equiv(m) \in Stable_k(T)$, and $m \in Stable_k(T) \Rightarrow equiv(m) \in Stable_i(T)$. Then, $SpuFree_i(T) = SpuFree_k(T)$.

Proof: In reducing $Stable_i(T)$ to $SigFree_i(T)$, P_i empties the $m.S$ field of every m in $Stable_i(T)$ and then discards duplicates. m and $equiv(m)$ are distinguished only by their signature fields, and setting this field to empty will make them identical. (See the definition of $equiv(m)$ in Section 4.1.) If $Stable_i(T)$ contains m and $equiv(m)$, the duplicate removal ensures that only one of the identical copies is retained in $SigFree_i(T)$. So, by the hypothesis of the lemma, $SigFree_i(T) = SigFree_k(T)$. Contrary to lemma, assume that $SpuFree_i(T) \neq SpuFree_k(T)$; also assume that, without loss of generality, there is an m such that $m \in SpuFree_i(T)$ and $m \notin SpuFree_k(T)$. By Definition 5.2, $SpuFree_i(T) \subseteq SigFree_i(T)$. We have established that $SigFree_i(T) = SigFree_k(T)$. So, $m \in SigFree_k(T)$. But $m \notin SpuFree_k(T)$; so, by Definition 5.2, there must have been an m' in $SigFree_k(T)$ such that $m.O = m'.O$. Since $SigFree_i(T) = SigFree_k(T)$, $m' \in SigFree_i(T)$. By Definition 5.2, m cannot be in $SpuFree_i(T)$. Hence the lemma. \square

Lemma 5.3: The protocol guarantees the unanimity condition, provided $d \geq \delta/(1 - 5\rho)$.

Proof: Consider an execution of the protocol in which P_i and P_k are non-faulty. Say, P_i accepts some m' . If m' is signed by P_k , then $equiv(m')$ is already accepted by P_k . If m' is not signed by P_k , then P_i will sign and diffuse m' to P_k . Since the timeliness bounds of Table 1 satisfy the unanimous acceptance requirement when $d \geq \delta/(1 - 5\rho)$, P_k will accept the diffused message. Thus, for every m' accepted by P_i , P_k accepts $equiv(m')$. Observe (from figure 2) that a non-faulty processor accepts no m , $m.TS \leq T$, once its stability counter (SC) becomes equal to T , $T \geq INIT_VAL$. So, for every m' accepted by P_i , P_k should accept $equiv(m')$ before

SC_k becomes equal to $m'.TS$. Therefore, for any $T \geq INIT_VAL$, when P_i and P_k form $Stable_i(T)$ and $Stable_k(T)$ respectively: $m \in Stable_i(T) \Rightarrow equiv(m) \in Stable_k(T)$.

By similar arguments, we can show: $m'' \in Stable_k(T) \Rightarrow equiv(m'') \in Stable_i(T)$. By Lemma 5.2, $SpuFree_i(T) = SpuFree_k(T) = SpuFree(T)$ (say) for every $T \geq INIT_VAL$. By Definition 5.2, $SpuFree(T)$ will contain at most one message originating from a given processor. Since processor ordering is unique and known, P_i and P_k will order the entries of $SpuFree(T)$ identically. Further, entries of $SpuFree(T)$ are ordered only after the entries of all $SpuFree(T')$, $T' < T$, have been ordered. So, P_i orders m' before m if and only if P_k orders m' before m . Thus the protocol satisfies the unanimity condition. \square

Theorem 5.1: The protocol guarantees unanimity and validity conditions with $\Delta = 4d(1 + \rho)$, provided $d \geq \delta/(1 - 5\rho)$.

Proof: Follows from Lemmas 5.1 and 5.3.

5.1. Optimal Upper Bound

We next show that the ordering bound of our protocol is the smallest achievable when (i) clocks are not synchronised, (ii) ordering is symmetric, and (iii) a processor cannot deduce temporal order between concurrent messages it receives. Each of these premises is defined below.

Unsynchronised Clocks: Non-faulty processors' clocks are not synchronised, where clock is a device which a processor uses for observing time. To state formally, let $c_i(\tau)$ denote the reading of P_i 's clock at real time τ . Clocks of non-faulty P_i and P_j are said to be unsynchronised during an interval ι if $|c_i(\tau) - c_j(\tau)|$ is arbitrary for every τ in ι .

Symmetric Ordering: Our protocol, like [20], is *symmetric* in the sense that the correct processors run the same program (except for process identities and signatures). In contrast, in an asymmetric protocol (e.g. [9]), correct processors can execute different code and hence play different roles: one processor, termed the sequencer, decides and disseminates the message ordering for other processors to accept what it has decided. In failure-free executions, asymmetric ordering will be the fastest if the non-sequencer processors can confirm the sequencer's correct behaviour without delaying message ordering; such delay-free confirmation is possible only when the sequencer is guaranteed to fail in a benign manner (e.g. by crashing). However, when (authenticated) Byzantine failures are permitted, the non-

sequencer processors must exchange messages to confirm that the sequencer had behaved correctly so far; that is, message diffusion must precede message ordering. Further, when the sequencer fails, message ordering is delayed until the failure is detected and a new sequencer is elected. Though not proved here, it appears that asymmetric ordering cannot offer a better worst-case ordering delay.

Non-deducibility of Temporal Order: Based on the definition of ‘happened before’ in [21], we define two messages to be *concurrent* if and only if neither one can be said to have happened before the other. Temporal order on messages is an order that is based on the Newtonian time instants at which messages were generated: for any two messages in the system, one message is before another in the temporal order if the first one was generated earlier in a Newtonian time-frame (see [22] for a formal definition). We assume that no processor can deduce temporal order between messages that are concurrent in the sense of [21]².

For simplicity we will assume $\rho = 0$ and consider a system in which communication delays between two non-faulty processors can be anything between 0 and δ_m , $0 < \delta_m < \delta$. Let δ_{m-} be a value such that $\delta_{m-} < \delta_m$ and $(\delta_m - \delta_{m-})$ is infinitely small.

Theorem 5.2: Any symmetric ordering protocol that works only with unsynchronised clocks, will have executions in which the ordering delay can be $3\delta_m + \delta_{m-}$.

Proof: By contradiction. Shown in Appendix B. □

By Theorem 5.2, the upper bound Δ on ordering delays must be at least $(3\delta_m + \delta_{m-})$, i.e. $\Delta \geq (3\delta_m + \delta_{m-})$. Since δ_m is not known directly, but only its upper bound δ (see assumption 4), $\Delta \geq 4\delta$. Theorem 5.1 establishes Δ of our protocol to be $4d(1 + \rho)$ with d recommended to be $d \geq \delta/(1 - 5\rho)$. When d is chosen to be $\delta/(1-5\rho)$, we have $\Delta \approx 4\delta$, if we assume $(1+\rho)/(1-5\rho) \approx 1$.

6. Comparison with Time-Based Approach

Let non-faulty processors’ clocks be synchronised within e : at any given real time instance τ , if a non-faulty processor’s synchronised clock reads T then any other non-faulty processor’s synchronised clock reads T' such that $T-e \leq T' \leq T+e$ where e is a *known* constant.

² According to [21], two messages need not be produced at the same newtonian time, for them to be deemed concurrent.

According to the classical time-based ordering protocol of [20], when (non-faulty) P_i forms and broadcasts a message m , it sets $m.TS$ to the current reading of its synchronised clock; P_i or any other non-faulty P_x stabilises an accepted m at its local synchronised time $m.TS + 2(d + e)$. So, if P_i has broadcast m at real time τ , then it orders m at real-time $\tau + 2(d + e)(1 + \rho_i)$. (To simplify the comparison of the two approaches, we will assume that the running rate of the synchronised clock of any non-faulty P_x is the same as that of P_x 's physical clock.) At τ , the synchronised clock of any non-faulty P_x , $x \neq i$, could be reading a value between $m.TS - e$ and $m.TS + e$. So, P_x will order m at some real time in the interval $[\tau + (2d + e)(1 + \rho_x), \tau + (2d + 3e)(1 + \rho_x)]$, depending on whether P_x 's synchronised clock is ahead of or behind P_i 's. So, in the best case when every non-faulty synchronised clock is ahead of the non-faulty transmitter's synchronised clock by e , the upper bound on time-based ordering delays Δ_{time} is maximum of $\{2(d + e)(1 + \rho_i), (2d + e)(1 + \rho_x)\}$. Thus, Δ_{time} is $2(d + e)(1 + \rho)$ in the best case and is $(2d + 3e)(1 + \rho)$ in the worst case.

The value of e depends on how frequently clocks are adjusted and on the algorithm used to compute the adjustment. The latter can be either an

- *external synchronisation algorithm* which requires a reliable time source (e.g. global positioning system (GPS)) to which processors are reliably connected [6], or an
- *internal synchronisation algorithm* which does not require any external assistance. It can be run entirely in hardware [5], or by software processes with specific hardware assistance as in MARS [23], or by software processes without any special hardware support.

In external, hardware based, or specialised-hardware assisted approaches, e obtained is very small (in the order of picoseconds) compared to d (which can be in the order of milliseconds). Thus, Δ_{time} is effectively $2d$, i.e., one half of the ordering delay Δ of the timeout approach. However, these approaches involve the use of specific devices or components which may not always be available to a system builder; in those circumstances, software implementation of an internal synchronisation algorithm using commercial, off-the-shelf (COTS) processors and operating systems is the only available option which we will focus on, in what follows.

Among the internal synchronisation protocols in the literature, the deterministic, and authenticated Byzantine fault-tolerant protocols of [24, 25, 26] are suitable for our TMR system and fault model. Among the rest which yield small e , some, such as [6], assume a

stronger fault model (of bounded omission failures), while others, such as [27] and the first two protocols in [25], assume a weaker fault model of unauthenticated Byzantine faults but are not appropriate for a TMR system as they require the system to have four processors.

In comparing Δ_{time} with Δ , we will make another simplifying assumption, favouring the time based approach: e is taken to be the maximum clock difference immediately after adjustments were made; i.e. we ignore a component of e which accounts for clock drift until next adjustment. Thus, the value of e turns out to be $\delta/(1+\rho)$ and $(1+5\rho)\delta/(1+\rho)$ when [24] and [26] are used to compute the adjustment respectively. (The authenticated protocol of [25] is considerably less efficient than [24].) In our calculations, we will take $e = \delta/(1+\rho)$ and, with no serious loss of accuracy, ignore the terms containing second and higher order powers of ρ ; this approximation will enable us to write, for example, $e = \delta/(1+\rho) = \delta(1-\rho)$.

$(\Delta_{\text{time}} - \Delta)$ for the best case of Δ_{time} becomes: $2(d + e)(1 + \rho) - 4d(1 + \rho) = 2(1 + \rho) [\delta(1+\rho) + \delta(1-\rho)] - 4(1 + \rho) [\delta(1+5\rho)]$. After some algebra, $(\Delta_{\text{time}} - \Delta) = -20\delta\rho$.

In the worst case for Δ_{time} , $(\Delta_{\text{time}} - \Delta) = -20\delta\rho + e(1 + \rho) = \delta(1-20\rho) \approx \delta$.

Note that the time-based protocol of [20], like ours, is symmetric and involves message diffusion and authentication. Hence the message complexity is the same for both.

We present two remarks over the evaluation of $(\Delta_{\text{time}} - \Delta)$ shown above. First, we have ignored a component of e which compensates the clock drift between successive adjustments. Since two non-faulty clocks can drift at the maximum rate of 2ρ , the missing component of e is $2\rho I$ where I is the period between successive adjustments. So, for example, if ρ is taken to be 10^{-6} and I is chosen to be 8.33 minutes, the value of e to be considered for comparison increases by 1 millisecond. Thus, the values of $(\Delta_{\text{time}} - \Delta)$ derived above hold in those contexts where the clocks are synchronised frequently often that $2\rho I$ remains negligibly small. Observe that frequent execution of clock synchronisation increases the message traffic, pushing the worst case delay δ for time-based protocol to a higher value.

Secondly, it is possible to reduce e by assigning a higher priority to clock synchronisation processes. This reduces the uncertainty in the delays for process scheduling and message queuing, thus resulting in a lower value for δ in $e = \delta/(1+\rho)$. It however has an implication on δ estimated for order protocol messages which now have a lower priority. Recall that the bound δ for order protocol messages is the maximum delay which non-faulty processors can possibly encounter during the entire system operation. This means that δ be estimated in the

most demanding scenario where the maximal set of higher priority processes are transmitting simultaneously. Thus, assigning a higher priority to synchronisation processes, while reducing δ for synchronisation messages, tends to increase δ for (time-based) order protocol messages.

6.1 Conditions Favouring Timeout Approach

We identify some favourable conditions in which our protocol works faster than the time based one. We define λ as the maximum difference within which non-faulty processors receive a given input from the environment and d_a as the actual maximum transmission delay that currently holds within the system. λ is typically called *tightness* [22] and will be small when inputs from the system environment are received via a broadcast LAN as shown in Figure 1; when the system is lightly loaded, d_a is much smaller than d (the worst-case estimate).

Suppose that all processors within the TMR system are non-faulty, and an input μ is first received by a processor at real-time τ . For simplicity, we will ignore the effect of non-zero ρ for the durations considered here. By $\tau + \lambda$, all processors must have formed and broadcast a message containing μ . By $\tau + \lambda + 2d_a$, P_i accepts two double-signed messages m' and m'' such that $path(m') = P_j:P_k$, $path(m'') = P_k:P_j$ and $m'.\mu = m''.\mu = \mu$; without loss of generality, let us assume: $m'.TS \leq m''.TS$. After $\tau + \lambda + 2d_a + 2d$, P_i will not accept

- (i) any double-signed m with $m.TS \leq m'.TS$ and $path(m) = P_j:P_k$ due to having accepted m' before $\tau + \lambda + 2d_a$ and due to the entry D3 of Table 1, and
- (ii) any double-signed m with $m.TS \leq m''.TS$ and $path(m) = P_k:P_j$ due to having accepted m'' before $\tau + \lambda + 2d_a$ and due to the entry E5 of Table 1.

Therefore m' , $m'.TS \leq m''.TS$, becomes stable at P_i by $\tau + \lambda + 2d_a + 2d$; i.e. every P_i orders input μ contained within m' by $\tau + \lambda + 2d_a + 2d$. When one of the processors is faulty, a similar reasoning indicates that non-faulty processors will order μ by $\tau + \lambda + d_a + 3d$.

With the time based protocol, non-faulty processors can order μ at or before $\tau + 2(d + e)$ and the explanation is as follows. Let P_i first receive the input (from the environment) at τ . The message formed and broadcast by P_i will be ordered by P_i at $\tau + 2(d + e)$ and by other processors at some time in $[\tau + (2d + e), \tau + (2d + 3e)]$. Assuming the best case (and in favour

of the time-based approach), we will regard that all other processors order at or before $\tau + (2d + e)$. Hence the maximum delay incurred for ordering the input is $2(d + e)$. Our protocol is guaranteed to order the inputs faster in the following situation:

- i. processors receive their inputs via a broadcast LAN which facilitates nearly simultaneous input arrival at all processors;
- ii. no special hardware support nor external assistance is available for synchronising clocks, leaving e comparable to d , the worst case communication delay envisaged;
- iii. the system is lightly loaded and message processing and queuing delays are small – leaving d_a and λ very small compared to d , such that $\lambda + 2d_a < 2e$ when no processor is faulty and $\lambda + d_a + d < 2e$ when one processor is faulty.

6.2. Implementation and Relative Performance

Estimation of d : Estimating d requires making an assumption on the maximum number of clients which can simultaneously send an input at a given time. We fixed this to be 10. To estimate d , a provisional estimate d_{pro} was first made as follows: 10 clients were made to issue their requests (to all three processors) at the same time. All messages were assumed to be timely and authentic, and therefore there was no message verification delay. The experiment ran until all clients had sent 1000 requests of 64 bytes. The maximum observed delay became d_{pro} which did not include the delay component due to maintaining path counters and performing timeliness checks on received messages. In the next stage, the experiment was repeated, using d_{pro} as the value for d , for both time and timeout based protocols, but now the protocols ran according to their complete description. The final value of d was chosen to be 10% more than the maximum observed delay, to account for inputs of size larger than 64 bytes.

Experimental Set-up: Since the clocks were synchronised frequently, e was taken to be d itself. We measured the *input ordering delay (IOD)* as the interval from the instance an input is first received by a non-faulty processor to the instant when at least two non-faulty processors are known to have ordered a message containing that input. To measure the average *IOD*, we ran a set of experiments in which one client sent a batch of 10 requests 100 times sequentially. (This emulates the situation of 10 clients accessing the system at different times.) Processors processed an ordered input by echoing it back to the client with a sequence number which should be identical for non-faulty processors.

The experiments were carried out in two different settings: the TMR processors were (i) T800 Inmos transputers [28] connected directly to each other by fast links (as shown in fig.1), and (ii) Pentium II 233 MHz PC's with 64MB of memory running the Linux 2.2.14 operating system and connected by a 100Mbps/sec fast ethernet. Thus, we consider two architectures commonly used for inter-processor communication: point-to-point and bus based. Further, the second implementation was done using Java (on Linux) – in a multithreaded environment where thread scheduling is a source of non-determinism which should not be allowed to affect the deterministic behaviour of replicas. We ensured this by implementing the order process – which makes the ordered delivery of stable messages - in a single thread.

Experimental Results: With Transputers, the values of d and λ were 50 ms (milliseconds) and 13.15 ms, respectively (and $e = d$). With all processors being non-faulty, the average values were: $d_a = 9.18$ ms; the IOD for the timeout protocol, $IOD_{To} = 136.28$ ms, and that for the time-based, $IOD_{Time} = 202.83$ ms. With one processor crashed, $d_a = 5.87$ ms, $IOD_{To} = 163.88$ ms, and $IOD_{Time} = 202.75$ ms. Observe that when the crashed processor is not participating in the protocol, d_a reduces but little change occurs to IOD_{Time} . The reduction in d_a is due to the reduced number of messages to be handled when one processor crashes: when there is no failure, each processor has to handle, for each client request, five messages (one directly from the client and two sets of two equivalent messages with each set originating from a given co-processor); this figure drops to two when one processor is crashed. Thus, a single crash results in 60% drop in the number of messages to be handled by an operational processor. (See [29, 18] for more experiments in the Transputer context.)

In the second, bus-based experimental set-up, the value chosen for d was 121 ms and λ was observed to be 11 ms. 99% of the delays observed were in the range [90, 105] ms when there were 10 clients, and in the range [60, 75] for a single client. The frequently-encountered delays being smaller than the chosen d helps the timeout protocol to be faster than the time-based protocol, as the Table below shows.

Clients	IOD_{To} (No failure)	IOD_{To} (One crash)	IOD_{Time} (No failure)	IOD_{Time} (One crash)
1	387	388	501	501
2	391	394	501	500
3	380	390	500	501
4	388	393	500	500
5	390	396	500	500
10	451	457	500	500

Table 2. Input ordering delays in a bus-based environment.

Observe that IOD_{To} increases only very slightly when a processor is crashed. Let us use subscripts 0 and 1 to differentiate the estimates made when no processor and one processor is crashed, respectively. So, $IOD_{To,0} = \lambda + 2d_{a,0} + 2d$. and $IOD_{To,1} = \lambda + d_{a,1} + 3d$. Recall that when a processor is crashed, the number of messages handled by a working processor drops by 60%. In a bus-based system, this also means a reduction in processors competing for bus access which often leads to a non-linear reduction in message transmission delays. Our measurements suggest that $(2d_{a,0} - d_{a,1}) \approx d$. Finally, the small difference of IOD_{Time} over the theoretical estimate (of $2(d + e) = 4d$) is attributed to the cost of thread scheduling in Java for delivering the ordered messages to the application process.

7. Related Work and Concluding Remarks

To our knowledge, the problem of message ordering in a distributed context was first addressed by Lamport [21]. We owe our use of message counters (*MCs*) in our protocol to his paper. The protocol in [21] is not fault-tolerant and is for an asynchronous context where any estimated bounds on processing, scheduling, and communication delays can be violated. For an equivalent problem of reaching agreement, Pease et. al. [1] provided synchronous time-based protocols for the least restrictive fault models of unauthenticated and authenticated Byzantine faults; they also showed that at least four processors are needed to contain one faulty processor when authentication is not employed. Among the works that ensued in the synchronous and time-based context, the following results are significant: protocols of [24, 25, 26] made internal clock synchronisation possible even for Byzantine fault models; even in the absence of faults, e cannot be guaranteed to be less than one half of the worst case delays expected for synchronisation messages [7]; reaching agreement simultaneously by non-faulty processors, necessary for identical message ordering, requires at least $(f+1)$ rounds of message diffusion if f is the maximum number of faults expected [16]. Cristian et. al. [20] proposed a suite of ordering protocols for a range of fault models, the weakest being the authenticated Byzantine.

In the domain of synchronous timeout approach, [9] and AMp of [10] are worth mentioning. The former is an *asymmetric* protocol and assumes, like us, an authenticated Byzantine model within a TMR system; further, it assumes every client to be a TMR system as well and solves the problem of message ordering together with majority voting of inputs. AMp was developed with commercial applications in mind, and provides the same message ordering guarantees as our protocol in a general n -processor system but assumes a benign

fault model where processors either crash or occasionally omit to produce responses. Our assumption of authenticated Byzantine faults is weaker and, as argued in [1], any further weakening of our fault model makes the desired form of message ordering impossible in a three-processor system.

In the asynchronous model, the processing, the scheduling, and communication delays are only known to be finite but their (upper) bounds cannot be known with certainty. Consequently, no deterministic message ordering protocol can be guaranteed to terminate even if one processor can crash [30]. This impossibility stems from the inherent difficulty in determining whether a remote processor has crashed or is only very slow. That is, since the asynchronous model permits any prior estimates of bounds to be violated, a fault-tolerant deterministic protocol cannot be guaranteed to terminate. It can only guarantee correctness without liveness: if non-faulty processes order a given message, they do so identically. This is in contrast to synchronous protocols which require the bounds to be inviolable; if violations undermine the fault hypothesis, a synchronous protocol will terminate within the guaranteed time period, but can cause non-faulty processors to order messages differently. Stating a precise set of requirements for eventual termination, asynchronous protocols such as [31] and [32] solve the ordering problem for non-Byzantine fault models, and [17] for the authenticated Byzantine model. These requirements generally warrant the violations of the assumed bounds to be below a threshold for a sufficiently long time.

In this paper, we have developed a synchronous timeout-based, ordering protocol for a TMR system. It borrows its structure from its time-based counterpart (thereby having the same message complexity) and replaces the synchronised time base with Lamport's logical clocks [21] and unsynchronised physical clocks. Where it required considerable design effort was in reducing the timeliness bounds by carefully analysing the various paths through which a processor can receive a message. This made the ordering delays smaller and the protocol optimal among the symmetric, timeout based protocols. The reductions achieved made use of the structure of the TMR system: there are only three processors of which at most one can fail. It is not clear whether the conservative bounds can be sufficiently reduced in a general system of n processors with at most f faults, to yield a smaller worst-case delay. This remains an open problem.

We have identified the contexts in which our protocol performs worse/better than the time-based protocol. When the synchronisation accuracy achieved is very small compared to the worst case message communication delays, time based approach is twice as fast as ours.

Where these two are nearly equal, as would be the case when clocks are synchronised with no assistance from special hardware or an external time source, our protocol offers a choice. We have identified analytically, and justified experimentally, the conditions in which our protocol orders inputs faster. Such conditions hold when the prevailing conditions of message traffic, failure and failure type within the system are less, or less severe, than the worst case conditions envisaged for the system. For example, as our experiments show, when the number of clients simultaneously accessing the system is less than the expected, the actual communication delays are certainly less than the estimated bound; similarly, no processor may fail for a considerable part of the TMR operation and even the failure occurred may be of type crash instead of the supposed Byzantine. Thus, in addition to deriving a timeout based protocol, the paper presents a practical alternative to time-based protocol when the achievable clock-synchronisation accuracy is comparable to the worst-case message communication delay envisaged.

Acknowledgements

This work has been supported in part by grants from CNPq/Brazil. Thanks to the anonymous reviewers for their constructive criticisms and suggestions.

References

- [1] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, 27(2), pp. 228-234, April 1980.
- [2] D. Powell, P. Verissimo, G. Bonn, F. Waeselync, and D. Seaton, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *Digest of Papers, FTCS-18, Tokyo*, pp. 246-251, June 1988.
- [3] F. B. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial," *ACM Computing Surveys*, 22(4), pp. 299-319, December 1990.
- [4] 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.
- [5] N. Vasanthavada and P. N. Marinos, "Synchronisation of Fault-Tolerant Clocks in the Presence of Malicious Failures," *IEEE Transactions on Computers*, Vol. C-37(4), pp. 440-448, April 1988.
- [6] P. Verissimo, L. Rodrigues, and A. Casimoro, "Cesium Spray: A Precise and Accurate Global Clock Service of Large Scale Systems," *Journal of Real Time Systems*, 11(3), 1997.
- [7] D. Dolev, J. Halpern, and H. R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronisation," *Proceedings of the 16th Annual ACM STOC*, pp. 504-511, April 1984.

- [8] 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.
- [9] K. Echtle, "Fault Masking and Sequence Agreement by a Voting Protocol with low Message Number," Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems, pp. 149-160, March 1987.
- [10] P. Verissimo, L. Rodrigues, and J. Rufino, "The Atomic Multicast Protocol (AMp)," In *Delta-4: A Generic Architecture for Dependable Distributed Computing*, D. Powell (Ed.) ESPRIT Research Papers, Springer-Verlag, pp. 267-294, 1991.
- [11] P. Verissimo, "Causal Delivery Protocols in Real-Time Systems: a Generic Model," Journal of Real Time Systems, 10(1), pp. 45-73, 1996.
- [12] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," Communications of the ACM, 31(2), pp. 120-126, February 1978.
- [13] P. D. Ezhilchelvan, "Early stopping algorithms for distributed agreement under fail-stop, omission, and timing fault types", In 6th Symposium on Reliability in Distributed Software and Database Systems, pages 201--212, Williamsburg, VA, March 1987.
- [14] D. Dolev, R. Reischuk, and H.R. Strong. "Early Stopping in Byzantine Agreement", Journal of the ACM, 37(4):720--741, October 1990.
- [15] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully, "Principle Features of the Voltan Family of Reliable System Architectures for Distributed Systems," IEEE Transactions on Computers, 41(5), pp. 542-549, May 1992.
- [16] D. Dolev and H. R. Strong, "Requirements for Agreement in a Distributed System," Proceedings of the 2nd Symposium on Distributed Databases, pp. 115-129, September 1982.
- [17] M Castro and B Liskov, 'Practical Byzantine Fault Tolerance', 3rd ACM Symposium on Operating Systems Design and Implementation (OSDI), pp. 173-186, Feb. 1999.
- [18] F. V. Brasileiro, P. D. Ezhilchelvan and N. A. Speirs, "TMR processing without explicit clock synchronisation," Proceedings of the 14th Symposium on Reliable Distributed Systems, Bad Neuenahr, Germany, pp. 186-195, September 1995.
- [20] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement," Digest of Papers, FTCS-15, Ann Arbor, USA, pp. 200-206, June 1985.
- [21] L. Lamport, "Time, Clocks, and Ordering of Events in a Distributed System," Communications of the ACM, 21(7), pp. 558-565, July 1978.

- [22] P. Verissimo and M. Raynal, "Time in Distributed System Models and Algorithms," in 'Advances in Distributed Systems', S. Krakowiak and S. K. Shrivastava (Eds.) LNCS 1752, Springer-Verlag, pp. 1-32, 2000.
- [23] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", IEEE Micro, pp. 25-41, 1989.
- [24] 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.
- [25] L. Lamport and P.M. Melliar-Smith, "Synchronising Clocks in the Presence of Faults," JACM 32(1), pp. 52-78, January 1985.
- [26] T.K. Srikant and S. Toueg, "Optimal Clock Synchronisation," Proceedings of the 4th ACM symposium on Principles of Distributed Computing, pp. 71-86, August 1985.
- [27] H.Kopetz and W. Ochsenreiter, "Clock Synchronisation in Distributed real time systems", IEEE Transactions on Computers, Vol C-36 (8), pp. 933-940, 1987.
- [28] INMOS Limited, Transputer Instruction Set, Prentice Hall International (UK) Ltd, 1988, ISBN 0-13-929100-8.
- [29] N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan, and S.K Shrivastava, "The Design and Implementation of Voltan Fault-Tolerant Systems for Distributed Systems, Transputer Communications, 1(2), pp. 1-17, November 1993.
- [30] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," Journal of the ACM, Vol. 32, No. 2, pp. 374-382, April 1985.
- [31] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", Journal of the ACM, 43(2), pp. 225 - 267, March 1996.
- [32] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", IEEE Transactions on Parallel and Distributed Systems, 10(6), pp. 642-657, June 1999.

Appendix A. Timeliness Bound C3

In Section 4.3.2, we claimed that the entry C3 in Table 1 can be $3d$ (instead of the conservative bound $4d$). We here show this by first proving the following lemma.

Lemma A1: 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_1 timely and 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. (See figure A1.) 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 be 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. \square

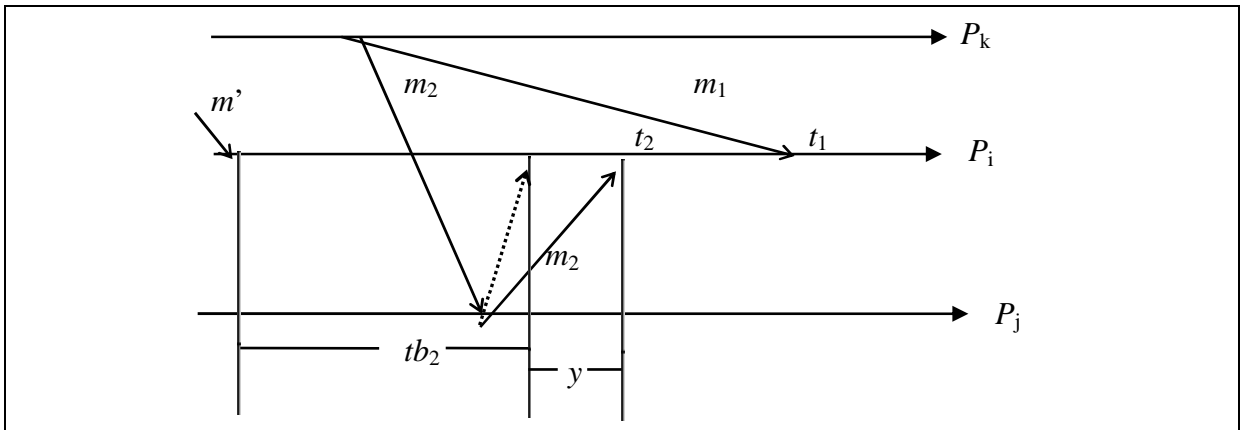
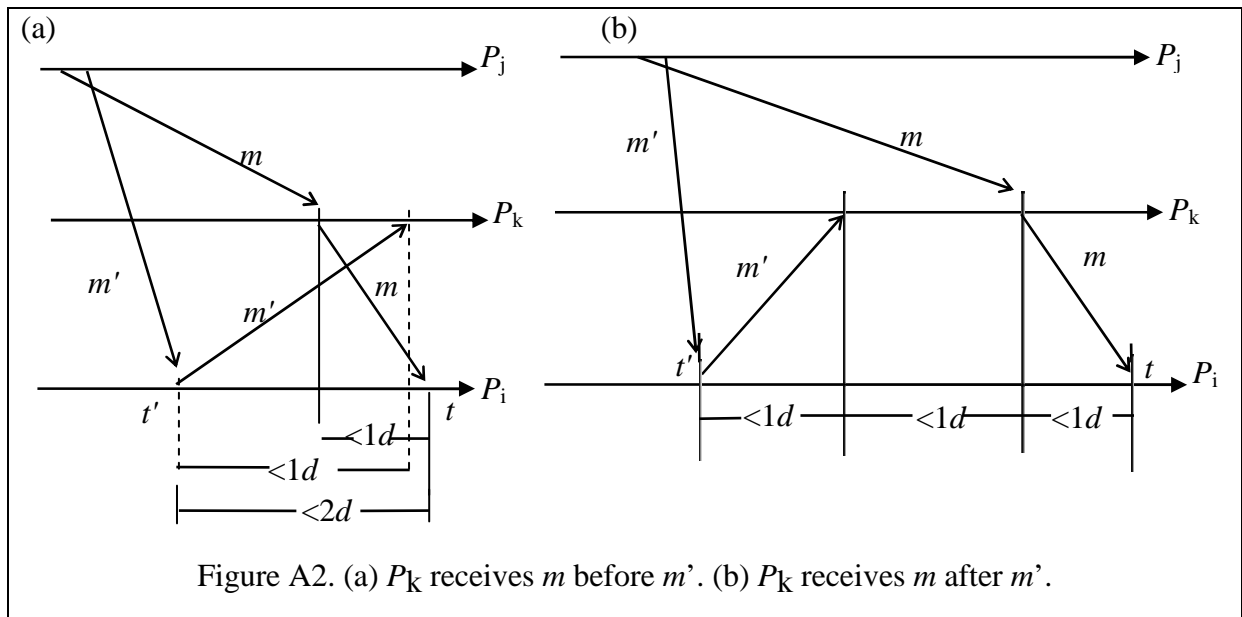


Figure A1. $m_1.TS \leq m_2.TS \leq m'.TS$.

Claim: The entry C3 of Table 1 is correct.

Proof: By given, $path(m') = P_j$, $path(m) = P_j:P_k$ and $m'.TS \geq m.TS$. Observe that P_k here is the non-faulty immediate sender of m and P_j can be faulty or non-faulty. (If P_j is non-faulty then $m'.TS > m.TS$.) Since P_i accepts m' (by hypothesis), it will diffuse the message to P_k . P_k also diffuses m to P_i . So, there are two cases to consider: P_k receives the single-signed m from P_j either (i) before or (ii) after it receives the diffused m' from P_i . The sub-case (i) is shown in figure A2(a). P_k can receive the diffused m' from P_i at any time before $t' + d$. Even if it diffuses m just before receiving m' , P_i can receive the diffused m just before $t' + d + d$; that is, $(t - t') < 2d$.

Figure A2(b) illustrates the second sub-case where P_k receives the single-signed m from P_j after it has received the double-signed m' from P_i . Since P_k diffuses m to P_i , it must have found the single-signed m it received as timely. By Lemma A1, P_k must find the double-signed m' timely. When it receives the single-signed m after having accepted the double-signed m' , it is in the same situation as P_i in case E1 where $path(m') = P_j:P_k$, $path(m) = P_j$ and the timeliness bound is shown to be d in Section 4.3.2. So, for the non-faulty P_k to find the single-signed m timely, it must have received m within d time after it received m' true; that is, the time elapsed between receiving m' and m must be less than d . From Figure A2(b), $(t - t')$ cannot be more than $3d$. Choosing the largest of the bounds estimated for the two sub-cases, $3d$ becomes the timeliness bound for the entry C3.



Appendix B. Protocol Optimality

Theorem 5.2: Any symmetric ordering protocol that works only with unsynchronised clocks, will have executions in which the ordering delay can be $3\delta_m + \delta_m$.

Proof: By contradiction. Assume that there is such a protocol which guarantees that ordering delays are *always* smaller than $3\delta_m + \delta_m$. Consider two distinct executions of this protocol during real-time intervals ι_1 and ι_2 respectively. By hypothesis, non-faulty processors' clocks remain unsynchronised throughout each interval. (Note: By this, we exclude a class of protocols which permit clock synchronisation messages to be piggybacked onto the order protocol messages and thus achieve clock synchronisation *during* the order protocol execution.)

In the first execution (see figure B1(a)), P_i fails *only* by not sending its messages to P_j and not receiving P_j 's messages. P_i sends m_i at its clock time t_i . Let m_i take zero time to reach P_k . Suppose that P_k 's clock reads t_k when P_k receives m_i . (Since m_i takes zero time, when P_k 's clock reads t_k , P_i 's clock reads t_i .) Let P_k accept and diffuse m_i to P_j and the diffused message take δ_m time to be received by P_j . Just before P_j receives the diffused m_i , i.e. when P_k 's clock reads $t_k + \delta_m$, suppose that P_j 's clock reads t_j and that P_j forms and sends m_j which takes δ_m time to be received by P_k . So, P_k 's clock reads $(t_k + \delta_m + \delta_m)$ when P_k receives m_j . Note that P_j sends m_j before it receives the diffused m_i from P_k ; therefore, neither m_i nor m_j happened before [21] the other. Since P_k cannot deduce that m_i originated before m_j in real-time, we will assume (without loss of generality) that m_j is ordered before m_i by all non-faulty processors.

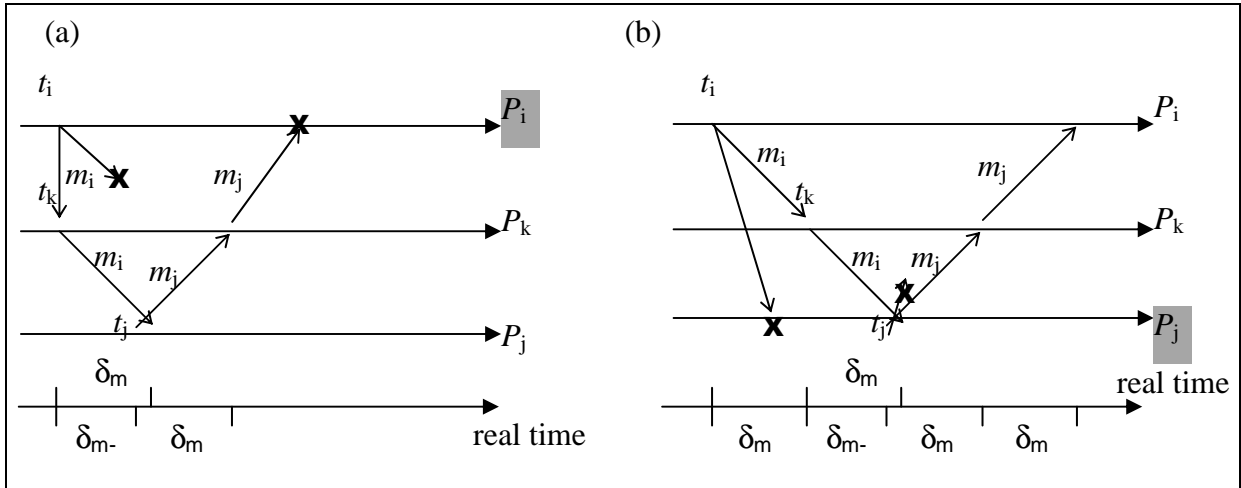


Figure B1. Execution Scenarios. (a) First execution. (b) Second execution.

In the second execution (see figure B1(b)), P_j fails *only* by not sending its messages to P_i and not receiving P_i 's messages. P_i sends m_i at its clock time t_i . m_i takes δ_m time to reach P_k and is not received by P_j . Suppose that P_k 's clock reads t_k when P_k receives m_i , i.e., when P_i 's clock reads $t_i + \delta_m$. (Note: this is possible with unsynchronised clocks whose readings can differ by an arbitrary amount.) Assume that the m_i diffused by P_k takes δ_m time to be received by P_j .

When P_k 's clock reads $t_k + \delta_{m-}$, suppose that P_j 's clock reads t_j and that P_j forms and sends m_j only to P_k which takes δ_m time to be received. That is, P_k 's clock reads $(t_k + \delta_{m-} + \delta_m)$ when P_k receives m_j . For P_k , this execution is indistinguishable from the first one. (Note that since the difference between the readings of unsynchronised clocks can differ by an arbitrary amount, we have chosen the difference to be a convenient amount that allows the following claim to hold: the two executions are indistinguishable for P_k even if the sender of a message m timestamps m with the local send time.) So P_k must order m_j before m_i . Since ordering delays are always smaller than $3\delta_m + \delta_{m-}$, non-faulty P_i must order its own m_i before $t_i + 3\delta_m + \delta_{m-}$. Say P_k 's diffused m_j takes δ_m time to be received by P_i . That is, P_i can receive m_j (for the first time), only when its clock reads $t_i + 3\delta_m + \delta_{m-}$. So, P_i can receive m_j only at or after $t_i + 3\delta_m + \delta_{m-}$. Hence P_i cannot order m_j before $t_i + 3\delta_m + \delta_{m-}$, and therefore before m_i . This violates the unanimity condition. This is a contradiction. \square

Appendix C: Timeliness Bounds and the Unanimous Acceptance Requirement

We here show that the timeliness bounds satisfy the unanimous acceptance requirement of section 4.2 when $d \geq \delta/(1 - 5\rho)$. This is done in two stages: lemma B1 proves that a non-faulty processor always finds another non-faulty processor's 1-signed message timely; and lemma B3 shows that when a non-faulty processor diffuses a 2-signed message to another non-faulty processor, the latter finds the diffused message timely. As in the main paper, we adopt the style of writing real time values in Greek and clock time values in italicised lower case Roman letters. The term 'clock' always refers to a processor's physical clock.

We assume the following notations: $START_i(p, \geq T)$ denotes the smallest real time instance when $PC_i[p]$ for path p becomes larger than or equal to T , $T \geq INIT_VAL$. That is, just before real time $START_i(p, \geq T)$, $PC_i[p]$ is less than T . $END_i(\leq T)$ denotes the largest real time instance when MC_i is less than or equal to T . That is, just after real time $END_i(\leq T)$, MC_i is larger than T and P_i will not form and send any m , $m.T \leq T$. We also retain the notation ρ_i , $|\rho_i| \leq \rho$, to denote the rate with which the clock of a processor P_i drifts from real time.

Lemma B0: Say a non-faulty P_i accepts m at real time τ_i . If m is not signed by a non-faulty P_k , P_k then receives m or $equiv(m)$ at real time τ_k such that $\tau_k < \tau_i + \delta$.

Proof: If m is P_i 's own message, P_i is required to send m to both P_j and P_k . If m is signed by P_j , P_i is required to diffuse double-signed $equiv(m)$ to P_k . By assumption 4, P_k then receives m or $equiv(m)$ before $\tau_i + \delta$. □

Lemma B1: When a non-faulty P_k forms and sends m , another non-faulty P_i finds m timely, provided $d \geq \delta/(1 - \rho)$.

Proof: By lemma B0, P_i receives m or $equiv(m)$ within δ real-time after P_k sends m . Say, $m.TS = T$. If P_i has not accepted any message with timestamp larger than or equal to T until it receives m , then $MC_i \leq T$ when it receives m and P_i finds m timely. Let us suppose that MC_i is already larger than T when P_i receives m . Since P_i has accepted one or more messages with timestamp larger than or equal to T , there must exist a message, say m' , $m'.TS \geq T$, whose acceptance causes P_i to set $PC_i[P_k] \geq T$ at real time $START_i(P_k, \geq T)$. If we show that

$START_i(P_k, \geq T) - END_k(\leq T) \geq \delta$ for every possible $path(m')$, then the lemma is proved. We present the proof by categorising the values of $path(m')$ in two cases. For each case, we construct the proof in the following manner. We define τ_i to be the real time when P_i accepts m' and $\alpha = START_i(P_k, \geq T) - \tau_i$. (Figure B1 shows these values along the real-time axis for P_i .) We derive two inequalities – inequality (1) involving τ_i and $END_k(\leq T)$ and inequality (2) involving τ_i and $START_i(P_k, \geq T)$. We then relate these two inequalities through the common element τ_i , to show what is required.

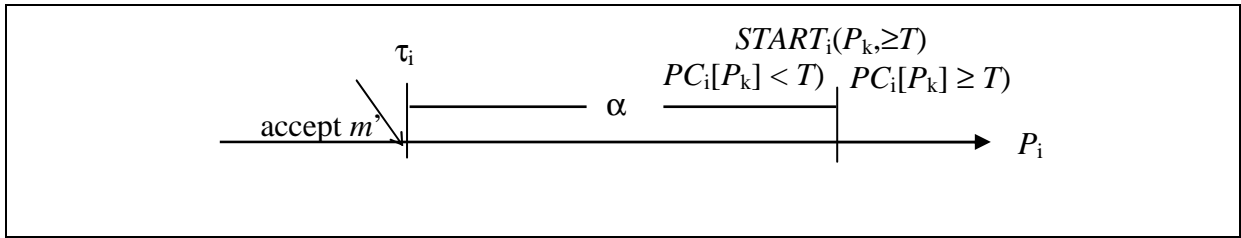


Figure B1. P_i accepting m' at τ_i sets $PC_i(P_k) \geq T$ at $START_i(P_k, \geq T)$.

Case I. $path(m') = P_k$ or $P_k:P_j$ or $P_j:P_k$. Just before sending m' , $m'.TS \geq T$, P_k sets MC_k to $m'.TS + 1$. Therefore,

$$END_k(\leq T) \leq \tau_i \quad (\text{I.1}).$$

For all the considered values of $path(m')$, $\alpha = d(1 + \rho_i)$. (See entries B1, E1 and D1 of table 1.) So,

$$START_i(P_k, \geq T) = \tau_i + d(1 + \rho_i) \quad (\text{I.2}).$$

Thus, combining (I.1) and (I.2) we have

$$START_i(P_k, \geq T) \geq END_k(\leq T) + d(1 + \rho_i).$$

Since $d \geq \delta/(1 - \rho)$ implies $d(1 + \rho_i) \geq \delta$, we have

$$START_i(P_k, \geq T) - END_k(\leq T) \geq \delta.$$

Case II. $path(m') = P_j$ or P_i . Let P_k receive m' (when $path(m') = P_i$) or $equiv(m')$ (when $path(m') = P_j$) from P_i , at real time τ_k . By lemma B0, $\tau_k < \tau_i + \delta$. Just before τ_k , we have either $MC_k \leq T$ or $MC_k > T$. In the first case $END_k(\leq T) = \tau_k$, and in the latter case $END_k(\leq T) < \tau_k$. So,

$$END_k(\leq T) - \delta < \tau_i \quad (\text{II.1}).$$

From entries A1 and C1 of table 1, $\alpha = 2d(1 + \rho_i)$. So,

$$START_i(P_k, \geq T) = \tau_i + 2d(1 + \rho_i) \quad (II.2).$$

Combining (II.1) and (II.2) we have

$$START_i(P_k, \geq T) > END_k(\leq T) - \delta + 2d(1 + \rho_i).$$

When $d \geq \delta/(1 - \rho)$, $2d(1 + \rho_i) \geq 2\delta$, thus

$$START_i(P_k, \geq T) - END_k(\leq T) \geq \delta. \quad \square$$

Lemma B2: $START_i(P_j; P_k, \geq T) \geq START_i(P_j, \geq T)$ for a non-faulty P_i and $T \geq INIT_VAL$.

Proof: Let m' , $m'.TS \geq T$ be the message whose acceptance at time τ causes P_i to set $PC_i[P_j; P_k] \geq T$ at $START_i(P_j; P_k, \geq T)$. From table 1, $\tau \geq START_i(P_j; P_k, \geq T) - 2d(1 + \rho_i)$; also, from the protocol in figure 2, P_i will schedule an update of $PC_i[P_j]$ to at least $m'.TS$ at latest by $\tau + 2d(1 + \rho_i)$, thus $START_i(P_k, \geq T) \leq \tau + 2d(1 + \rho_i)$, and since $d > 0$, $START_i(P_j; P_k, \geq T) \geq START_i(P_k, \geq T)$. \square

Lemma B3: When a non-faulty P_i diffuses a double-signed m to non-faulty P_k , P_k finds m timely, provided $d \geq \delta/(1 - 5\rho)$.

Proof: As in lemma B1, we will let $m.TS = T$ and suppose that MC_k is already larger than T when P_k receives m . Since P_k has accepted message(s) with timestamp larger than or equal to T , there must exist a message, say m' , $m'.TS \geq T$, whose acceptance causes P_k to set $PC_k[P_j; P_i] \geq T$ at real time $START_k(P_j; P_i, \geq T)$. Let τ_k be the real time when P_k accepts m' and $\alpha = START_k(P_j; P_i, \geq T) - \tau_k$. Figure B2 shows these values along the real-time axis for P_k .

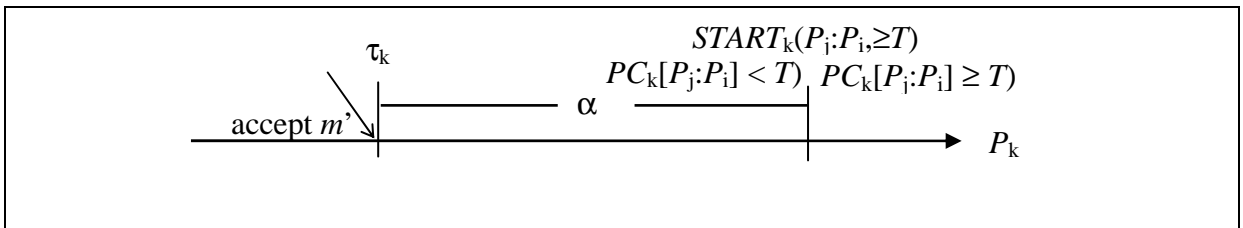


Figure B2. P_k accepting m' at τ_k sets $PC_k(P_j; P_i) \geq T$ at $START_k(P_j; P_i, \geq T)$.

Note that P_i must have diffused m to P_k before $START_i(P_j, \geq T)$, and that P_k will find m timely if it receives m before $START_k(P_j; P_i, \geq T)$. We need only to show that $START_k(P_j; P_i, \geq T) - START_i(P_j, \geq T) \geq \delta$ for every possible $path(m')$. We show this by

considering three cases regarding the *originator* of m' , $m'.O$. In all cases, there is a real time τ_1 when P_i either sends m' or receives $equiv(m')$; in the same way we did to prove lemma B1, for each case we construct two inequalities: inequality (3) involving τ_1 and $START_k(P_j:P_i, \geq T)$, and inequality 4 involving τ_1 and $START_i(P_j, \geq T)$. These equalities are then related through τ_1 to show what is required.

We first present the timeliness bounds that a non-faulty P_k uses to check the timeliness of a message m it receives. They are presented in the same way that we did before for a non-faulty P_i , and are obtained simply by interchanging the suffixes k and i in table 1. Table B1 shows the timeliness bounds used by P_k .

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

Table B1: P_k 's timeliness bounds for a past m given that MC_k first exceeded $m.TS$ due to accepting m' .

Case I. $m'.O = P_k$. From entry A3 of table B1, $\alpha = 4d(1 + \rho_k)$ and P_k must have formed and sent m' at real time $START_k(P_j:P_i, \geq T) - 4d(1 + \rho_k)$. Let P_i receive m' at τ_1 , by lemma B0,

$$\tau_1 < START_k(P_j:P_i, \geq T) - 4d(1 + \rho_k) + \delta \quad (\text{I.3}).$$

From lemma B1, P_i accepts m' ; so from entry B2 of table 1, we have

$$START_i(P_j, \geq T) \leq \tau_1 + 2d(1 + \rho_i) \quad (\text{I.4}).$$

Combining (I.3) and (I.4) we have

$$START_i(P_j, \geq T) < START_k(P_j:P_i, \geq T) - 4d(1 + \rho_k) + \delta + 2d(1 + \rho_i), \text{ and}$$

$$START_k(P_j:P_i, \geq T) - START_i(P_j, \geq T) > 2d(1 + 2\rho_k - \rho_i) - \delta$$

which leads to

$$START_k(P_j:P_i, \geq T) - START_i(P_j, \geq T) > \delta,$$

whenever $d \geq \delta/(1 + 2\rho_k - \rho_i)$, which is always true since $d \geq \delta/(1 - 5\rho) > \delta/(1 + 2\rho_k - \rho_i)$.

Case II. $m'.O = P_i$. Note that the m' that P_k accepts can be single or double-signed. Let P_i form and send m' (if m' is single-signed) or the single-signed $equiv(m')$ (if m' is double-signed) at real time τ_i . Whether m' is single or double-signed, the entries B3 and E3 of table B1 indicate that $\alpha = 3d(1 + \rho_k)$. Since $\tau_i \leq \tau_k$,

$$\tau_i \leq START_k(P_j:P_i \geq T) - 3d(1 + \rho_k) \quad (\text{II.3}).$$

From entry A2 of table 1

$$START_i(P_j \geq T) \leq \tau_i + 2d(1 + \rho_i) \quad (\text{II.4}).$$

Combining (II.3) and (II.4) we have

$$START_i(P_j \geq T) \leq START_k(P_j:P_i \geq T) - 3d(1 + \rho_k) + 2d(1 + \rho_i), \text{ and}$$

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) \geq d(1 + 3\rho_k - 2\rho_i)$$

which leads to

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) \geq \delta,$$

whenever $d \geq \delta/(1 + 3\rho_k - 2\rho_i)$, which is true since $d \geq \delta/(1 - 5\rho) \geq \delta/(1 + 3\rho_k - 2\rho_i)$.

Case III: $m'.O = P_j$. Say $path(m') = P_j$. From entry C3 of table B1, $\alpha = 3d(1 + \rho_k)$. Let P_i receive the diffused $equiv(m')$ from P_k at real time τ_i ; from lemma B0,

$$\tau_i < START_k(P_j:P_i \geq T) - 3d(1 + \rho_k) + \delta \quad (\text{III.3a}).$$

If P_i accepts $equiv(m')$ at τ_i , then from entry D2 of table 1, $\tau_i \geq START_i(P_j \geq T) - d(1 + \rho_i)$; otherwise, $\tau_i > START_i(P_j:P_k \geq T)$ and from lemma B2, $\tau_i > START_i(P_j \geq T)$. Since $d > 0$, in both cases we have

$$START_i(P_j \geq T) \leq \tau_i + d(1 + \rho_i) \quad (\text{III.4a}).$$

Combining (III.3a) and (III.4a), we get

$$START_i(P_j \geq T) < START_k(P_j:P_i \geq T) - 3d(1 + \rho_k) + \delta + d(1 + \rho_i), \text{ and}$$

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) > 2d(1 + 3\rho_k/2 - \rho_i/2) - \delta,$$

which leads to

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) > \delta,$$

whenever $d \geq \delta/(1 + 3\rho_k/2 - \rho_i/2)$, which is true since $d \geq \delta/(1 - 5\rho) > \delta/(1 + 3\rho_k/2 - \rho_i/2)$.

Say $path(m') = P_j:P_i$. $\alpha = 2d(1 + \rho_k)$ from entry D3 of table B1. Let P_i diffuse m' to P_k at real time τ_i . Since $\tau_i \leq \tau_k$,

$$\tau_i \leq START_k(P_j:P_i \geq T) - 2d(1 + \rho_k) \quad (\text{III.3b}).$$

From entry C2 of table 1,

$$START_i(P_j \geq T) \leq \tau_i + d(1 + \rho_i) \quad (\text{III.4b}).$$

Combining (III.3b) and (III.4b) we get

$$START_i(P_j \geq T) \leq START_k(P_j:P_i \geq T) - 2d(1 + \rho_k) + d(1 + \rho_i), \text{ and}$$

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) \geq d(1 + 2\rho_k - \rho_i),$$

which leads to

$$START_k(P_j:P_i \geq T) - START_i(P_j \geq T) \geq \delta,$$

whenever $d \geq \delta/(1 + 2\rho_k - \rho_i)$, which is true since $d \geq \delta/(1 - 5\rho) > \delta/(1 + 2\rho_k - \rho_i)$. \square