Bounds on the Time to Detect Failures Using Bounded-capacity Message Links

Stephen Ponzio*
MIT Laboratory for Computer Science

Abstract

We consider a system of distributed processors that communicate by passing messages and that have inexact information about time. Specifically, a processor knows that a single message is delayed by at most time d and the time between any two of its consecutive steps is at least c_1 and at most c_2; it has no other way of estimating elapsed time. This simple model is very close to traditional models used in distributed computing theory, and has been studied by Attiya and Lynch [2, 1] among others. We extend the model by making a realistic assumption about how the delay of messages is affected by the rate at which they are sent. We define a model of message links with ***bounded capacity***, which are guaranteed to deliver messages at only a given rate. If a processor sends messages at a greater rate, they may incur greater delay.

We quantify the effect of this bounded capacity on the time necessary to detect processor failures. We consider a system of two processors connected by a bi-directional message link of (integral) capacity μ. First we give two very simple protocols that guarantee any stopping failure will be detected within time $2Cd + d$ and $C^2d/\mu + Cd + d$ respectively, where $C = c_2/c_1$. The main result is an almost-matching lower bound of $2Cd+d/\mu$ or $C^2d/\mu+Cd+d$, whichever is less. If the link is uni-directional, our result specializes to give a matching upper and lower bound of $C^2d/\mu + Cd + d$.

*Supported by an NSF Graduate Fellowship

1 Introduction

In fault-tolerant distributed algorithms, a common primitive for detecting failures is to "time out" failed processors. If processors fail by simply stopping, then a failure may be detected by the absence of messages from a processor. We consider how quickly such failures can be detected in a semi-synchronous model where processors have inexact information about time, subject to realistic assumptions about message links.

We use a very high level of abstraction of message links; for our purposes, the "message link" consists of all processing at levels lower than the process or code that is actually implementing the timeout algorithm. The delay of a message is taken to be the total amount of time that elapses between the processor step at which the algorithm code specifies that a message should be sent and when the recipient processor reads in that message. At our level of abstraction, a message link is assumed to be completely reliable, delivering all messages in the order sent. We will assume that all messages are of some fixed length, and ignore the possible affect of a message's length on its delay.

For simplicity, we consider a system of two processors.[1] A processor may send messages only at atomic steps (which take zero time). It knows that the time between each pair of its consecutive steps is at least c_1 and at most c_2. These bounds hold for each processor and are common knowledge to each. A processor cannot tell directly how much time has elapsed between two particular steps—only that the

[1] In [1], where systems of n processors are considered, it is assumed that each pair of processors is connected by a private bidirectional message link. There, it is natural to assume that a timeout protocol executes independently for each pair of processors. If we measure "detection time" to be from failure until detection by *every* processor, the bounds hold without this assumption.

1052-8725/92 $3.00 © 1992 IEEE

elapsed time is within these bounds. For each model of message links considered, a message sent in the absence of other messages is always delivered within time d of when it is sent. We are interested in the worst-case time between when a processor fails and when the failure is detected by the other processor.

If it is assumed that *every* message is delivered within time d of when it is sent (regardless of message traffic—"unbounded capacity"), then the following simple protocol minimizes the time between any failure and its detection.[2] Each processor sends a message at every step that it takes. If a processor takes more than $(d + c_2)/c_1$ steps without receiving a message, it declares the other processor faulty. We see that non-failed processors are never declared faulty: the time between delivery of any two consecutive messages is at most $c_2 + d$ and at least this much time—$(d + c_2)/c_1$ steps, each taking at least time c_1—is waited by the recipient before declaring the sender faulty. The maximum time between a failure and its detection is approximately $Cd + d$, where $C = c_2/c_1$, occurring in the following scenario: processor p broadcasts a message at time t and then fails immediately; this message is delivered to q at time $t + d$; after receiving this message, processor q runs slowly (its steps separated by c_2) and thus p's failure is not detected until time $t + d + c_2((d + c_2)/c_1) = t + d + C(d + c_2)$. It is not difficult to show that this detection time is optimal.

Although the above protocol guarantees minimal delay between any failure and its detection, it requires sending messages nearly continuously. It relies on the strong assumption that all messages are delivered within time d, regardless of the rate at which they are sent. In reality, the rate at which a processor issues messages may be much greater than the rate at which messages may be actually sent and delivered. In such a case, lower-level processing buffers the messages by holding them in a queue in memory until they can actually be sent. If messages are continually issued at a rate greater than they can be sent, then the number of queued messages grows without bound. Thus there is an upper bound on the rate at which messages can be issued before such backup occurs. It is precisely this rate that will affect the bound on detection time. Our model gleans this rate by the following reasoning. The maximum total delay d of a single message may account for several levels of processing. It is therefore likely that although one message is sent at time t and delivered at time $t + d$, another message may be sent before time $t + d$ without affecting the total delay of the second message. That is, messages sent between the processors (algorithm code) may be pipelined to some degree. The number of messages that can be in transit between the processors without affecting the total delay of any message correponds roughly to our notion of the "capacity" μ of a message link. In our model, if all messages sent are separated by at least time d/μ, then each one is delivered within time d; if two messages are sent within less than time d/μ of each other, the second message may be delayed by more than d. This is the essential property of the message links we will consider. Our upper bounds depend only on this property; our lower bounds depend only on the property that if messages are sent at too great a rate then they become backed up and their delay grows.

We first give two very simple protocols that guarantee any stopping failure will be detected within time $2Cd+d$ and $C^2d/\mu+Cd+d$ respectively. The main result is an almost-matching lower bound of $2Cd + d/\mu$ or $C^2d/\mu + Cd + d$, whichever is less. Note that the first expression of the lower bound is not tight with the first expression of the upper bound. If the link is uni-directional, our result specializes to give a matching upper and lower bound of $C^2d/\mu + Cd + d$.

2 Model and definitions

Our underlying formal model is that proposed by Lynch and Attiya [2, 1], following the timed automata model of [5]. We consider a system of two processors, p and q. Each processor is a deterministic (possibly infinite) state machine. Formally, an execution of the algorithm is a sequence of *configurations* alternated with *events*. A configuration is a vector of the processors' local states. Each event has a real time associated with it and these times are non-decreasing with the sequence. Events are of three types:

1. A processor computation step. In a computation step, a processor, based on its local state, may perform local computation and send a finite number of messages to other processor(s).
2. A message delivery, $del(m, x)$. A message delivery event may denote the delivery of a message to a processor ($del(m, p)$) or to a message

[2] This is the strategy employed in the algorithms of [1] and [6], which assume such unbounded-capacity message links.

link ($del(m, \ell)$). If a message delivery event denotes the delivery of a message m to a processor p, p may change its local state according to m and its current state (i.e., "remember" the delivered message). Events corresponding to messages "delivered" to links serve only to mark the time at which a message is sent, as explained in the next section.

3. A processor failure. During a failure step, a processor's normal computation transition function is applied to its local state, but only a subset of the specified messages are actually sent and the processor goes into a permanent *failed* state. In all subsequent steps, it remains in the failed state.

For further details and precise formalism, we refer the reader to [1]. Our formalism for messages links captures their interaction with processors by considering a message m sent by p to q at time t to be "delivered" at time t to the message link connecting p and q. That is, we consider only executions in which every computation event at which a message m is sent is followed by a delivery event $del(m, \ell)$ with the *same* real time, where ℓ is the appropriate message link. If the message link has unbounded capacity (like those in [1, 6]), then valid executions must include an event no later than time $t + d$ corresponding to the delivery of m at q. For simplicity, we will assume that message links deliver messages in the order sent. Our algorithms do not make strong use of this assumption and our lower bounds hold in spite of it. The formalism for bounded links is described below. Also, we consider only executions in which the successive steps of all processors are separated by at least time c_1 and at most c_2. Note that a processor does not "know" directly the time between two particular steps—this quantity is not an argument to the state transition function. Although it is not necessary for the proof, we will generally assume that $c_2 \ll d$: processor step time is very small compared to the maximum message delay time. In the analysis, we therefore approximate the quantity $d + c_2$ by d.

2.1 Modeling bounded-capacity links

In modeling bounded-capacity message links, we would like to capture the reality that if messages are sent too frequently, they may take longer to be delivered.

We define a message link of *unit capacity* and *delay* d from p to q as follows: if p sends a message m to q at time t and t' is the time at which the previous message from p to q is delivered to q, then m is delivered to q by time $\max(t, t') + d$. If $t < t'$, then we say that the message link "queues" m while the previous message is "in transit" during the interval (t, t'). For positive integer μ, we define a message link of *capacity* μ and *delay* d as the serial composition of μ message links $\ell_1, \ldots, \ell_\mu$, each of unit capacity and delay d/μ. These links are connected serially so that messages are delivered from ℓ_i to link ℓ_{i+1} ($1 \leq i < \mu$) and ℓ_μ delivers messages to the recipient process. Formally, we consider only executions in which the computation event with the sending of m is followed by a $del(m, \ell_1)$ event at the same time, and every delivery event $del(m, \ell_i)$ is followed by a delivery event $del(m, \ell_{i+1})$ (or $del(m, p)$ if $i = \mu$) within time d/μ.

We note that this model does not necessarily correspond to physical reality, but is meant to capture the degree of message pipelining available between the highest level processors that are exchanging messages. It is this parameter, the number of messages that may be "in transit" at the same time without an increase in their delay, which affects the efficiency of failure detection.

Thus, in the absence of any other message traffic, the delay of a single message is bounded by $\mu \cdot d/\mu = d$. Note that if a single component link delays all messages by its maximum amount, d/μ, then messages are delivered at a maximum rate of μ messages per time d. In particular, it is easy to see that if the last component link delays each message by d/μ, then for any interval of time of length l, at most $\left\lceil \frac{l}{d/\mu} \right\rceil$ messages are delivered. If no two messages are sent within time d/μ of each other, then each message is delivered within time d of when it is sent. This is easily seen by induction. For the lower bound, we assume only that when a link delays each message by the maximum possible amount, at least time d/μ elapses between the delivery of messages. For this reason, our results hold for other models of message links with capacity μ and delay d, such as μ d-delay links of unit capacity in parallel, each connected directly with p and q.

2.2 Timing out failed processors

A processor is said to *detect the failure* of another processor when it irrevocably decides that the other

has failed. A timeout protocol is *correct* if it satisfies two properties for all executions and all processors p and q: (1) if p fails and q does not fail, then q eventually detects the failure of p, and (2) if neither p nor q fails, then neither p nor q detects the failure of the other.

For a given execution α, we say that p detects the failure of q *within time* T in α if q fails at time t in α and p detects the failure of q at time $t' \leq t + T$ in α,. We say a timeout protocol *guarantees a detection time of* T if for all processors p and q and all executions α in which p fails but q does not, q detects the failure of p within time T in α.

3 Simple upper bounds

An upper bound of $2Cd + d$ is achieved by a simple protocol that works for *any* link capacity. The two processors continually exchange a single "token" message: when p receives the token message from q, it sends the token message back to q, and q does likewise. If a processor takes more than $2(d + c_2)/c_1$ steps without receiving a message, it concludes that the other processor is faulty. Because there is at most one message in transit at any time, it is always delivered within time d of when it is sent. Clearly a nonfaulty processor is never timed out. This protocol guarantees that any failure is detected within time $2Cd + d$ (to be precise, $d + 2C(d + c_2) + c_2$; recall we approximate $d + c_2 \approx d$): if p fails at time t, then by time $t + d$ all of the messages it has sent to q are delivered and q has sent its last message to p; within another time $c_2(1 + 2(d + c_2)/c_1) \approx 2Cd$, q has taken enough steps to conclude that p has failed.

An upper bound of $C^2d/\mu + Cd + d$ is achieved by a one-way protocol in which each processor sends a message every $(d/\mu)/c_1$ steps. This is "one-way" in the sense that only messages from p to q are used to detect the failure of p and these messages are independent of messages from q to p. A processor concludes that the other has failed if it takes more than $(Cd/\mu + d)/c_1$ steps without receiving a message. Clearly, the sending times of every two messages are separated by at least time d/μ and therefore, as shown in Section 2.1, each message is delivered within time d of when it is sent. The maximum amount of time between the delivery of two consecutive messages from a given processor is $c_2(d/\mu)/c_1 + d = Cd/\mu + d$ (if the first message is delivered immediately, the sender runs slowly, and the following message incurs the maximum possible delay, d). This is less than the minimum amount of time, $Cd/\mu + d + c_1$, that the other processor waits before detecting failure. This protocol guarantees a detection time of $C^2d/\mu + Cd + d$: if p fails at time t, then by time $t + d$ all of the messages it has sent are delivered to q; within another time $c_2(Cd/\mu + d)/c_1 = C^2d/\mu + Cd$, q has taken enough steps to conclude that p has failed. Thus we obtain a simple upper bound of $\min(2Cd+d,\ C^2d/\mu+Cd+d)$.

4 The lower bound

We now prove a nearly corresponding lower bound of $\min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$. Note that $2Cd + d/\mu < C^2d/\mu + Cd + d$ if and only if $\mu < C + 1$. Thus, the bounds are tight except for $\mu < C + 1$; in particular, when $\mu < C$, $2Cd + d$ is the best upper bound and $2Cd + d/\mu$ is the best lower bound.

We first prove that there exists some execution in which p runs "fast" (its steps separated by time c_1), q runs "slowly" (its steps separated by time c_2), messages from q to p are delivered immediately, messages from p to q are delayed by at least time d, and at least time d/μ elapses between when p sends some pair of messages. We prove that such an execution is possible for any protocol that is guaranteed to detect failures within *any bounded amount of time*. This is proved below using the properties of the bounded-capacity message links. The idea is that if the last component link from p to q delays all messages by d/μ then the delivery of every pair of messages is separated by time d/μ. Therefore, if each pair of messages sent by p were separated by less than d/μ, then messages would be sent faster than they were delivered. Thus the number of messages sent but undelivered and, consequently, the total delay of a message, would grow in time without bound. After p crashes, the queued messages are still delivered to q, and q cannot tell that p has crashed until the queue is emptied.

Lemma 4.1 *For any correct timeout protocol that guarantees a bounded detection time, there exists an execution in which (1) all consecutive steps of p are separated by c_1, of q are separated by c_2; (2) all messages from q to p are delayed by time 0, from p to q are delayed by at least time d; and (3) for any t_0, there exists a pair of messages m_1 and m_2 sent by p at times t_1 and t_2 respectively, with no message*

sent by p in between those times, such that $t_1 \geq t_0$, $t_2 - t_1 \geq d/\mu$.

Figure 1 depicts an example of such an execution.

Proof: Fix any execution β of the protocol in which (i) the first three timing constraints are satisfied, (ii) each component link from p to q delays each message by time d/μ, and (iii) no processor fails. Such an execution exists because conditions (i), (ii) and (iii) are independent of each other and within the bounds of the model. Clearly, condition (ii) implies that all messages from p to q are delayed at least time d. We prove that third condition is also satisfied in β. To do so, assume for contradiction that it is not.

First note that because β is unbounded in length, p must send an unbounded number of messages: if it does not, then let m_ℓ be the last message that it sends and consider an execution γ in which p fails after sending m_ℓ. Because q receives the same messages from p in each execution, it cannot distinguish between the two executions and therefore q either does not detect p's failure in γ or erroneously decides that p has failed in β.

Recall that a processor can send messages only at steps and p's steps are separated by exactly time c_1 in β. It follows that if two consecutive messages are not separated by at least time d/μ, then they are separated by at most $k = \left\lceil \frac{d/\mu}{c_1} \right\rceil - 1$ steps, which is time $k \cdot c_1 < d/\mu$.

Consider the interval $[t_0, t_0 + x]$ of execution β, where x is defined below. Because p sends an unbounded number of messages and, by assumption, every two consecutive messages are separated at most time kc_1, processor p sends at least $\lfloor x/(kc_1) \rfloor$ messages in this interval. But since the last component link delays each message by d/μ, at most $\left\lceil \frac{x}{d/\mu} \right\rceil$ messages are delivered in this interval. Thus the number of messages sent but not delivered in this interval is at least $(\frac{x}{kc_1} - 1) - (\frac{x}{d/\mu} + 1)$. According to the properties of the message links, the last message sent in this interval may not be delivered until all prior messages have been delivered. Thus the last message sent by p in this interval may not be delivered until time $t_0 + x + \frac{d}{\mu}(\frac{x}{kc_1} - \frac{x}{d/\mu} - 2)$. If B is the bound on detection time guaranteed by the protocol, then let x be large enough so that $\frac{d}{\mu} \cdot (\frac{x}{kc_1} - \frac{x}{d/\mu} - 2) > B$ (recall that $kc_1 < d/\mu$).

We conclude that the last message sent by p in the interval $[t_0, t_0 + x]$ of β is not delivered until after time $t_0 + x + B$. Since p does not fail in β, q does not time out p; in particular, q does not time out p before time $t_0 + x + B$. However, before time $t_0 + x + B$, this execution is indistinguishable to q from an execution in which p fails at time $t_0 + x$ and which is otherwise identical to β at p and q up to times $t_0 + x$ and $t_0 + x + B$, respectively. Therefore in this execution q does not detect the failure of p within time B. Thus the protocol cannot be correct; this contradiction proves the lemma. ∎

Our lower bound proof uses the retiming techniques of "shifting" events in time and "shrinking" portions of executions that were used in [2] and [4]. The basic strategy of the proof is as follows. Beginning with an execution β given by Lemma 4.1, we know that if p were to fail during the step at which it sends m_1, then q would declare p faulty by time $t_1 + T$. We would like to show that if the bound T guaranteed by the algorithm is small, then there is an execution in which q declares p faulty though p has not failed. Notice that q cannot tell if p has failed or not until message m_2 arrives or doesn't. The idea for showing such an execution exists is to ask, what if instead, all events at q occurred earlier by d and messages from p to q arrived earlier by d? Because q does not have a way of telling time absolutely, it's cannot tell the difference between this execution and β. (Formally, it's state transition function takes as input only the current state and possibly a delivered message and so its sequence of states is the same in the two executions.) We say that the two executions are *indistinguishable* to q. Using a similar reasoning, we further ask, what if p ran slowly (its steps separated by c_2) between the when it sends m_1 and when it sends m_2, and q runs fast (its steps separated by c_1) after it receives m_1, and m_2 is delayed by the maximum possible amount, d? We are able to show that in the resulting execution, q declares p faulty too soon—that is, before m_2 arrives.

We will use the technique above several times, where, given an interval of events in which a processor is running slowly (time c_2 between its steps), we create a new execution in which that processor runs fast over that interval of events (time c_1 between its steps). We call this construction "shrinking" that interval. Conversely, given an interval in which a processor is running fast, we may create a new execution in which we "stretch" that interval. Of course, we preserve the order of events at that processor by accordingly retiming message delivery events at that processor, and we must verify that the timing con-

straints on message delivery are not violated. This may require individually retiming the message delivery events at the component message links $\ell_2 \ldots \ell_\mu$. This is easily done within the time bounds of each individual delivery event, as long as the total delay of any message is not increased to be greater than d or made less than 0. We will suppress the detail of retiming the delivery events at the component links and verify only that the total delay of any message is within the proper bounds.

Theorem 4.2 *In a system with links of capacity μ and delay d, no correct timeout protocol can guarantee failures to be detected within less than time* $\min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$.

Proof: Let $T = \min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$. For contradiction, assume the existence of a protocol that guarantees a detection time of T. We do not make use of the particular value of T until the final step of the proof (the construction of execution β''). We will reach a contradiction by showing that there is an execution of the protocol in which p does not fail but q decides that it has.

Let β be an execution of the protocol whose existence is implied by Lemma 4.1 with $t_0 = d\left(\frac{C}{C-1}\right)$. Let m_1 and m_2 be the two messages specified by the lemma, sent by p at times t_1 and t_2 respectively. Figure 1 depicts an example of an execution satisfying Lemma 4.1; for presentation, messages from p to q are shown taking exactly time d, and messages from q to p are shown sent at arbitrary times.

Let α be an execution in which (i) events at p are identical to those of β up to time t_1, (ii) p fails at time t_1 after sending m_1, and (iii) events at q are identical to those of β up to time $t_1 + d/\mu + d$. Clearly α exists, since in β, message m_2 does not arrive until $t_2 + d \geq t_1 + d/\mu + d$ and thus the state transitions of q in α are identical those in β until this time. Also, the assumed protocol guarantees that in α, q detects the failure of p before time $t_1 + T$.

The rest of the proof proceeds as follows. By shifting the events of q in α and β, we construct executions α' and β', which are indistinguishable from α and β respectively, to both p and q. By retiming the events of α', we construct α'', which to q is indistinguishable from α'. By retiming the events of β', we construct β'', which to q is indistinguishable from α'' until the event at which it times out p. Execution β'' is also indistinguishable to p from β' until after it sends m_2. Thus, although p does not fail in β'', q times out p in β'', contradicting the correctness of the assumed protocol.

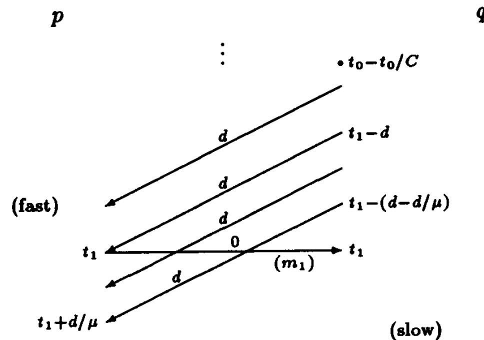

Figure 1: In the region of interest, execution α' is simply α with events of processor q occurring earlier in time by d. Because p fails at time t_1, q detects the failure of p by time $t_1 + T - d$, denoted by the circle.

4.1 Construction of α' and β'

Conceptually, we wish to construct α' from α by letting each event at q occur earlier in time by d ("shifting" those events earlier by d).[3] Figure 1 depicts the suffix of α', showing the shifted events of the region in which we shall be interested.

This execution satisfies the timing constraints on message delivery, since messages sent by p (delayed

[3] Strictly speaking, this may not be possible for all events at q because of initial conditions. However, because we are really only interested in shifting the events in the region around t_1, we may "shrink" some earlier interval of the execution, so that events subsequent to that interval occur earlier by time d, as desired. In particular, we may shrink (by a factor $c_2/c_1 = C$) the interval $[0, \frac{C}{C-1}d]$ of α, so that it corresponds to the interval $[0, \frac{1}{C-1}d]$ of α'. Thus the last event of this interval is shifted earlier by $\frac{C}{C-1}d - \frac{1}{C-1}d = d$. Recall that we chose $t_0 = \frac{C}{C-1}d$ in β and $t_0 \leq t_1$, so in β, the last event of this interval occurs before time t_1, when m_1 is sent; in β', the last event of this interval occurs at $t_0 - d$, before time $t_1 - d$.

by at least d in α) are received by q at most d earlier in α' and hence are delayed by at least 0 in α'; messages sent by q (delayed by 0 in α) are sent at most d earlier in α' and hence are delayed by at most d in α'.

Execution β' is constructed similarly, shifting earlier by d the events at q in β.

Because p and q do not know the time between any particular pair of steps they take, they cannot distinguish between either α and α' or β and β'. It follows that α' and β' are not distinguishable to p up to the point at which it fails and not distinguishable to q up to when it receives m_2 in β' (which is at least time $t_2 \geq t_1 + d/\mu$). Also, q's detection of p's failure occurs before time $t_1 + T - d$ in α'.

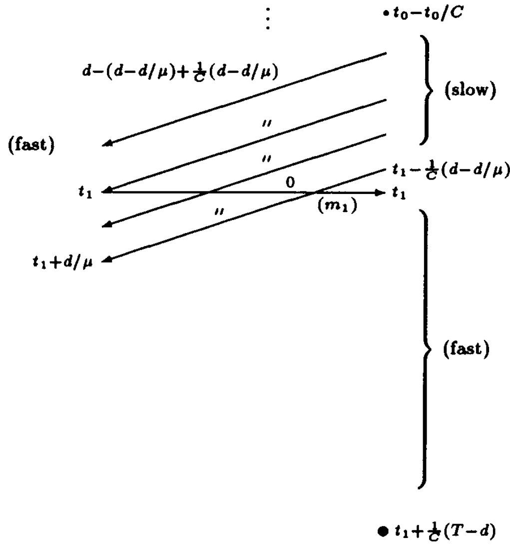

Figure 2: Execution α'' is constructed from α' by mapping the interval $[t_1-(d-d/\mu),\ t_1+(T-d)]$ of α' to the interval $[t_1 - \frac{1}{C}(d-d/\mu),\ t_1 + \frac{1}{C}(T-d)]$ of α'' and appropriately shifting the rest of q's events.

4.2 Construction of execution α''

Recall that q runs slowly in α and α'—its steps are separated by c_2. We now construct α'' from α' by retiming certain events at q. Events at p are the same as in α' up to time t_1, when p fails in both executions; events at p after time t_1 are inconsequential to the proof and may be defined arbitrarily within the bounds of the model.

The retiming operation at q maps the interval $[t_1 - (d-d/\mu),\ t_1+(T-d)]$ of α' to the interval $[t_1 - \frac{1}{C}(d-d/\mu),\ t_1 + \frac{1}{C}(T-d)]$ of α'' by letting q run fast over this interval in α''. The mapping shrinks the events around time t_1: events at time t_1 in α' also occur at t_1 in α''; events in the above interval of α' are retimed to occur closer to time t_1 by a factor of C. The rest of execution α'—before time $t_1-(d-d/\mu)$ and after time $t+(T-d)$—is shifted to preserve the step times of events on the borders of this interval. To be precise, α'' is defined at q by retiming each event that occurs at q at time $t' \geq \frac{1}{C-1}d$ in α' to occur at q at time t'' in α'', where t'' is defined as follows:

$$t'' = \begin{cases} t' + (d-d/\mu) - \frac{1}{C}(d-d/\mu) & \text{shift} \\ \quad \text{if} \quad \frac{1}{C-1}d \leq t' \leq t_1 - (d-d/\mu) & \\ t_1 + \frac{1}{C}(t'-t_1) & \text{shrink} \\ \quad \text{if} \quad t_1 - (d-d/\mu) \leq t' \leq t_1 + (T-d) & \\ t' - (T-d) + \frac{1}{C}(T-d) & \text{shift} \\ \quad \text{if} \quad t' \geq t_1 + (T-d) & \end{cases}$$

This execution is illustrated in Figure 2.[4]

By construction, this retiming operation does not cause violations of the bounds on processor step times. We now verify that α'' is consistent with the timing assumptions for message delivery. First note that all events at p before time t_1 occur at the same time in executions β, α, α' and α''. We show that for any event at q occurring at time t'' in α'' and at time t in α (and hence at $t' = t - d$ in α') such that $t'' \leq t_1$ and $t \geq \frac{C}{C-1}d$, we have $t - d \leq t'' < t$. By the retiming mapping above, if $t' < t_1$, then

$$t' \leq t'' \leq t' + (d - d/\mu) - \frac{1}{C}(d - d/\mu),$$

(this is because t' is mapped forward in time furthest when $t' \leq t_1 - (d-d/\mu)$; least when $t' = t_1$) which,

[4] Again, we need to shift the events before time $t_1 - (d-d/\mu)$ while preserving initial conditions. To do this we partially undo the shrinking performed on the interval $[0, \frac{C}{C-1}d]$ of α. These events were mapped to the interval $[0, \frac{1}{C-1}d]$ of α', in which q runs fast, with the last event of the interval occurring exactly time d earlier in α' than in α. In α'', we need the last event of this interval to occur exactly time $(d-d/\mu) - \frac{1}{C}(d-d/\mu)$ later than in α'. Because this amount is less than d, we are able to do this, in effect partially undoing the original shrinking. The timing assumptions for steps of q are clearly satisfied. Because the net effect from both shrinking operations is to shift any particular event in the interval $[0, \frac{C}{C-1}d]$ of α earlier by less than d in α'', the timing assumptions for message delivery are also clearly satisfied, for the reasons outlined in the discussion of α'.

substituting $t' = t - d$, gives

$$t - d \leq t'' < t - d/\mu - \frac{1}{C}(d - d/\mu) < t. \quad (1)$$

In α, every message from p to q is delayed by at least d. We claim that in α'', every message from p to q is delayed by at least time 0 and by less time than in α. If a message is delivered at q after time t_1 in α'', then because p sends no messages after time t_1, it must be sent by t_1 (no new message receipts at q have been introduced to α'') and hence delayed at least time 0; also, events at q after time t_1 in α'' occur earlier in α'' than in α, so the message is delayed by less than it is in α. If a message is delivered at q at time $t'' \leq t_1$ in α'' then by Equation (1), it is delivered earlier in α'' than in α by not more than d; because this message is delayed by at least d in α, it follows that it is delayed by at least time 0 in α''.

We also claim that the delay of each message from q to p in α'' is delayed by at least 0 and at most d. In α, all messages from q to p are delayed 0; if in α'' they are sent before t_1, then from Equation 1 they are sent earlier (and delayed more) by not more than d. The receipt of any message sent by q after t_1 is inconsequential to the proof and is defined arbitrarily to be within the bounds of the model.

Finally, we note that q detects the failure of p before time $t_1 + \frac{1}{C}(T - d)$ in α''.

4.3 Construction of execution β''

We now construct execution β'' in which p does not fail and which is indistinguishable to q from α'' up to time $t_1 + \frac{1}{C}(T - d)$. In proving that β'' satisfies the timing assumptions on step time and message delivery, we will, for the first and only time, make use of the fact that $T = \min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$. Because q times out p before time $t_1 + \frac{1}{C}(T - d)$ in α'', we conclude that in β'', q mistakenly times out the nonfaulty p, contradicting the assumed correctness and completing the proof.

To construct β'' at q we use exactly the same events as in α'', up to time $t_1 + \frac{1}{C}(T - d)$. We do not specify the events occurring at q later than this except to say that any message sent by p after time t_1 is delayed by time d.

At p, we construct β'' from β' by mapping the interval $[t_1,\ t_1 + d/\mu]$ of β' to the interval $[t_1,\ t_1 + \frac{1}{C}(T - d) - d]$ of β'' (p runs fast over this inteval in β'; it runs more slowly over this interval in β''). Events in this interval are retimed to occur further

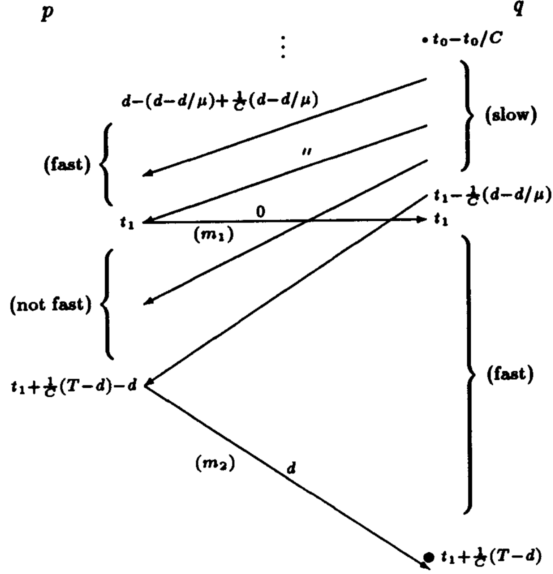

Figure 3: Execution β'' is essentially the same as execution α'', except that p does not fail; instead, it runs slowly after sending message m_1, and message m_2 is delayed by d. Because p sends no other messages before m_2, this execution appears the same as α'' to q until it receives m_2.

from time t_1 by at most a factor of C (as we will show). We do not specify events occurring at p after time $t_1 + \frac{1}{C}(T - d) - d$ except to say that any message sent by q after time $t_1 - \frac{1}{C}(d - d/\mu)$ is delivered at p exactly time d later. Thus, β'' is defined at p (up to $t_1 + \frac{1}{C}(T - d) - d$) by retiming each event that occurs at time t' in β' to occur at time t'' in β'', where t'' is defined as follows:

$$t'' = \begin{cases} t' & \text{if} \quad t' \leq t_1 \\ t_1 + \frac{\frac{1}{C}(T-d)-d}{d/\mu}(t' - t_1) & \text{if} \quad t_1 \leq t' \leq t_1 + d/\mu \end{cases}$$

This execution is illustrated in Figure 3.

We now verify that β'' is consistent with the timing assumptions of the model. Note that all events of β'' at p before time t_1 are the same as in $\beta, \beta', \alpha, \alpha'$, and α''; events of β'' at q before time $t_1 + \frac{1}{C}(T - d)$ are by definition the same as in α''. Having already verified the timing properties for α'', we need verify only the timing properties involving events (processor steps, message sends, and message receipts) occurring at p in the interval $[t_1, t_1 + \frac{1}{C}(t - d) - d]$. Events occurring at p later than $t_1 + \frac{1}{C}(T - d) - d$ and at q later than $t_1 + \frac{1}{C}(T - d)$ are inconsequential to the proof and may

be scheduled in any way consistent with the bounds of the model. Basically, we just need to ensure that we are not causing p to run too fast over this interval (that's why we need $T \leq C^2d/\mu + Cd + d$) and that messages from q to p are not delivered more than d after they are sent (that's why we need $T \leq 2Cd+d$).

First, we verify that successive steps of p after t_1 are separated by at most c_2. We show that for any interval $[t''_i, t''_j]$ of β'', mapped from the interval $[t'_i, t'_j]$ of β', where $t_1 \leq t'_i \leq t'_j \leq t_1 + d/\mu$, we have $t''_j - t''_i \leq C(t'_j - t'_i)$:

$$\begin{aligned} t''_j - t''_i &= (t'_j - t'_i)\frac{\frac{1}{C}(T-d) - d}{d/\mu} \\ &\leq (t'_j - t'_i)\frac{\frac{1}{C}(C^2d/\mu + Cd) - d}{d/\mu} \\ &= (t'_j - t'_i)C. \end{aligned}$$

Because any two steps of p are separated by time c_1 in β', they are separated by at most $C \cdot c_1 = c_2$ in β''.

We now verify that the delays of messages sent by p after t_1 are within the proper bounds. Actually, it turns out that m_2 (the first message sent by p after t_1) is sent by p after time $t_1 + \frac{1}{C}(T-d) - d$: m_2 is sent at $t_2 \geq t_1 + d/\mu$ in β' (and β) and thus at $t''_2 \geq t_1 + \frac{1}{C}(T-d) - d$ in β''. Messages sent by p after time t_1 are specified to be delayed by at least time d, so m_2 is not delivered until at least time $t_1 + \frac{1}{C}(T-d)$ (after q times out p). The delivery of m_2 and all subsequent messages by p is consistent with our definition of β'' at q.

We now verify that messages from q to p are within the proper bounds. We analyze these messages in three cases according to when they are sent by q in execution β' (which is the same time as they are sent in α').

Case 1: q sends at time $t' \leq t_1 - d$ in β' (and α'). These messages are delivered to p by time t_1 in β' (in *beta*, they were sent at t_1 and delivered immediately). Events at p before t_1 in β' occur at the same time in β''. These events occur at the same time in α' and α'' (events at p before t_1 were not changed in going from α' to α''). Since both the send events and receive events for these messages occur at the same times respectively in β'' and α'', the analysis of α'' shows that their delays are within the proper bounds.

Case 2: q sends at time t' in β' (and α'), where $t_1 - d \leq t' \leq t_1 - (d - d/\mu)$.
In β'' (and α'') the sending event at q is shifted to time $t' + (d - d/\mu) - \frac{1}{C}(d - d/\mu)$, which is less than t_1. Such a message is delivered at time $t' + d$ in β' where $t_1 \leq t' + d \leq t_1 + d/\mu$. In β'' the delivery event at p is mapped to $t_1 + \frac{1}{d/\mu}(\frac{1}{C}(T-d) - d)(t' + d - t_1)$, which is greater than t_1. Thus, such a message is properly delivered after it is sent (delayed by at least 0). With some calculation, using the definitions of T and t' and noting $\frac{1}{d/\mu}(\frac{1}{C}(T-d) - d) - 1 \geq 0$, it is easily verified that the difference between $t_1 + \frac{1}{d/\mu}(\frac{1}{C}(T-d) - d)(t' + d - t_1)$ and $t' + (d - d/\mu) - \frac{1}{C}(d - d/\mu)$ is at most d.

Case 3: q sends at time $t' \geq t_1 - (d - d/\mu)$ in β'. These messages are sent at $t'' > t_1 - \frac{1}{C}(d - d/\mu)$ in β'' and thus are defined to be delivered at p exactly time d later. Note that such messages are delivered at p later than time

$$\begin{aligned} t_1 - \frac{1}{C}d + \frac{1}{C}d/\mu + d &= t_1 + 2d + \frac{1}{C}d/\mu - \frac{1}{C}d - d \\ &= t_1 + \frac{1}{C}(2Cd + d/\mu - d) - d \\ &\geq t_1 + \frac{1}{C}(T-d) - d. \end{aligned}$$

This is consistent with our definition of β'' at p.

Thus we conclude that β'' is a valid timed execution in which p does not fail but q times out p. This is a contradiction on the correctness of the assumed protocol. ■

4.4 Bounds for a unidirectional message link

We remark that our proof gives a tight upper and lower bound of $C^2d/\mu + Cd + d$ for a system of two processors with a message link in only one direction.

In such a system, we have two processors, p and q, and a single message link of capacity μ from p to q. Naturally, a protocol does not need to detect failures of q. All other previous definitions apply.

The second simple protocol described in Section 3 operates independently in each direction. It immediately gives a protocol for the unidirectional case, guaranteeing that in any execution, q detects the failure of p within time $C^2d/\mu + Cd + d$.

It is also not difficult to see that our lower bound proof of Theorem 4.2 specializes to the unidirectional case to give a corresponding lower bound of $C^2d/\mu + Cd + d$. Theorem 4.2 is proved for $T = \min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$. A similar theorem for the unidirectional case may be proved with $T = C^2d/\mu + Cd + d$. Recall that in that proof, the value of the timeout detection time T guaranteed

by the protocol is not used before the claims about execution β''. All preceding claims except those involving messages from q to p carry over a fortiori. Lemma 4.1, for example, is true also for the unidirectional case with the exception of its third condition, which regards messages from q to p. The proof of Theorem 4.2 uses the fact that $T \leq 2Cd + d/\mu$ in claims about β'' only to verify bounds on the delay of messages from q to p. This analysis is not needed for a theorem about the unidirectional case and hence the entire proof specializes to the unidirectional case to give a lower bound of $C^2d/\mu + Cd + d$.

5 Conclusion

We have attempted to reconcile an important assumption of [2, 1, 6] with the reality that processors can, but should not, send messages faster than they can be delivered. Our model intended to capture the parameters that are important to the designer of algorithms for failure detection: the total message delay time and the amount of pipelining that is available within our entire (high-level) message link abstraction. Our formal model of bounded-capacity links was motivated not by a desire to reflect physical reality but to quantify this degree of pipelining. Using this model, we were able to derive mathematical results that depend explicity on these parameters.

Acknowledgments

Thanks to Greg Troxel and Nancy Lynch for helpful discussions.

References

[1] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. Report TM-435, Laboratory for Computer Science, MIT, November 1990. Also in STOC 1991.

[2] H. Attiya and N. A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Proc. 10th IEEE Real-Time Systems Symposium*, 1989, pp. 268–284. Also: Technical Memo MIT/LCS/TM-403, Laboratory for Computer Science, MIT, July 1989.

[3] M. Fischer, N. Lynch and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol. 32, No. 2 (1985), pp. 374–382.

[4] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, Vol. 62, Nos. 2/3 (August/September 1984), pp. 190–204.

[5] M. Merritt, F. Modugno and M. Tuttle. Time constrained automata. *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, 1991.

[6] S. Ponzio. Consensus in the presence of timing uncertainty: omission and Byzantine failures. *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 1991, pp. 125–138. Also: The real-time cost of timing uncertainty: consensus and failure detection. MIT SM Thesis, June 1991. Available as MIT Lab. for Computer Science Technical Report MIT/LCS/TR-518, October 1991.