The Real-Time Cost of Timing Uncertainty: Consensus and Failure Detection

by

Stephen J. Ponzio

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering
and
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1991

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
May, 1991

Certified by ______________________________

Nancy A. Lynch
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

The Real-Time Cost of Timing Uncertainty: Consensus and Failure Detection

by

Stephen J. Ponzio

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1991 in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Electrical Engineering and
Master of Science in Electrical Engineering and Computer Science.

Thesis Supervisor: Nancy A. Lynch
Title: Professor of Computer Science

Keywords: consensus, agreement, timing uncertainty, semi-synchrony, partial synchrony, fault-tolerance, Byzantine failures, omission failures, timeout.

Abstract

In real distributed systems, processes may have only inexact information about the amount of real time needed for primitive operations such as process steps. This thesis studies the effect of this timing uncertainty on the real-time behavior of distributed systems. We consider a semi-synchronous model in which the amount of real time between process steps is known to be in the interval $[c_1, c_2]$ and every message is known to be delivered within time d of when it is sent. We use $C = c_2/c_1$ as a measure of the timing uncertainty.

We first study the problem of reaching agreement in the presence of failures. A simple argument derived from the case of synchronous processes shows that at least time $(f+1)d$ is required to tolerate f failures, while time $(f+1)Cd$ is sufficient to tolerate f stopping or omission failures by directly simulating the rounds of any synchronous consensus algorithm. We narrow this gap for omission failures, building on the nearly optimal algorithm of Attiya, Dwork, Lynch, and Stockmeyer which tolerates only stopping failures. If fewer than half the processes are faulty ($n \geq 2f+1$), then the running time of our algorithm is $4(f+1)d + Cd$, which is within a factor of 4 of optimal and has minimal dependency on the timing uncertainty factor C. If more than half the processes are faulty, then a more complicated analysis shows the running time is increased by approximately a factor of $\min(\frac{f}{n-f}, \sqrt{C})$. We also present a general simulation for $n \geq 3f+1$ tolerant of Byzantine failures that simulates any synchronous algorithm at a cost of time $2Cd + d$ per round.

Finally, motivated by the message inefficiency of our consensus algorithm for omission failures, we define a more realistic model of message links by limiting their capacity. If messages are sent too frequently on these message links, they may incur delay greater than d. For message links with capacity μ, we prove nearly tight upper and lower bounds of $\min(2Cd + d,\ C^2d/\mu + Cd + d)$ and $\min(2Cd + d/\mu,\ C^2d/\mu + Cd + d)$ respectively for the time needed to detect stopping failures.

Acknowledgments

It is a pleasure to thank many friends and colleagues for their support and assistance during the research and writing of this thesis.

My advisor Nancy Lynch has been an invaluable source of encouragement and enthusiasm; I am grateful to her for having made this thesis possible. Cynthia Dwork taught me much about the consensus problem; many hours of discussion with her have been enjoyable and enlightening. Hagit Attiya suggested numbering messages in the omissions algorithm. Baruch Awerbuch helped with the proof of Lemma 3.23. John Leo provided useful comments on earlier versions of Chapters 3 and 4.

Thanks also to my friends for making the past few years more enjoyable, particularly Bob Struble, Brett S. Lowry, Michael Klugerman, Dave Doyno and Judy Marlinski. Finally, I thank my parents and family for their love.

This research was done at the MIT Laboratory for Computer Science with the support of a National Science Foundation Graduate Fellowship and support from the Office for Naval Research under contract numbers N00014-85-K-0168 and N00014-91-J-1046, the National Science Foundation under contract numbers CCR-8611442 and CCR-8915206, and the Defense Advanced Research Projects Agency under contract numbers N00014-89-J-1988 and N00014-87-K-0825.

Contents

Chapter 1

Introduction

In real distributed systems, processes are likely to be neither perfectly synchronous nor completely asynchronous. Many systems lie somewhere between these two extremes and can thus be more accurately modeled by a *semi-synchronous* model in which processes have inexact knowledge about real time. In our model, the degree of asychrony is captured by a parameter which we call the processes' *timing uncertainty.* We will be particularly interested in how the magnitude of timing uncertainty affects the time complexity of distributed computing problems. In particular, we study the the time needed to reach consensus in the presence of omission failures and in the presence of Byzantine failures. We also introduce a model of message links with bounded-capacity and study the time needed to detect failures in a system using these message links.

In a synchronous system, processors have perfectly synchronized clocks and distributed algorithms are often broken up into rounds of communication. In a single round of communication, each processor may receive messages from other processors, perform some local computation, and then send messages to other processors. The time required to perform local operations is generally assumed to be negligible and the time complexity of algorithms is therefore measured by the number of rounds of communication required. In an asynchronous system, the delay of messages is arbitrary and unbounded (or the relative rates of different processors are unbounded). The time complexity of an asynchronous algorithm is usually measured by letting one time unit equal the maximum delay of any message ([Gal82, Awe85]).

The model we use is a slightly simplified version of the semi-synchronous model introduced in [AL89], which is in turn based on the formal model of timed automata in [MMT90]. In this model, processors have inexact knowledge about the time needed to perform certain primitive operations. The model is formally described in Section 2.1, but is very simple: every message is delivered within time d of when it is sent and the amount of time between any two consecutive steps of any process is in the interval $[c_1, c_2]$. Because process steps are the only events for which there is a lower bound, a process can deduce a lower bound on the amount of time for any interval of events only by counting the number of steps it takes in that interval. For instance, to ensure that time d elapses over an interval of events, a processor must count d/c_1 of its local steps, after these events it knows that at least time $c_1 \cdot d/c_1 = d$ (and at most $c_2 \cdot d/c_1$) has elapsed. We will be particularly interested in how this timing uncertainty factor of c_2/c_1, henceforth denoted C, affects the time complexity of problems relative to their synchronous time complexity.

Of particular interest are problems that are intractable in an asynchronous setting yet have solutions with tight bounds in the synchronous setting. A simple example is the basic task of detecting the failure of stopped processes. Clearly, if there is no bound on message delay or relative process step time, then failures can never be detected with certainty; in a synchronous system, any stopping failure can be detected within approximately the maximum message delay time. Another natural candidate is the consensus problem. It is well known that a completely asynchronous algorithm for consensus cannot tolerate the failure of even one process, whereas exactly $f+1$ rounds of synchronous communication are needed to tolerate f failures in a synchronous system.

1.1 Reaching consensus—known time bounds

The problem of reaching consensus in the presence of failures is one of the most well-studied problems in distributed computing. We consider the version of this problem for a system of n deterministic processes some f of which may fail, completely connected by a reliable message system. The processes begin executing at the same time, each with a private binary input, and must each decide on a binary value such that no two nonfaulty processes decide differently and if all processes begin with value v then v is the decision of all nonfaulty processes. In this thesis, we consider two kinds of process failure: send-omission failures, by which a process may unwittingly omit messages of an algorithm, and Byzantine failures, by which a process may exhibit arbitrary behavior.

It is well known ([FLP85]) that in an asynchronous system, this problem cannot be solved deteministically even if the only failure to be tolerated is the unannounced halting (stopping) of a single process. The work of [DDS87] methodically explores the synchrony necessary to reach consensus; they show that if there is no upper bound on message delay or there is no upper bound on the relative rate of process steps—if any of our bounds d, c_1, or c_2 does not hold—then there is no deterministic solution tolerating even a single stopping failure.

The time complexity of the consensus problem has been well studied in the synchronous rounds model (see, for example, [LSP82, PSL80, FL82, DS83, DLM82]). It is well known that $f+1$ rounds of communication are both sufficient ([PSL80]) and necessary ([FL82, M85, DM86, CD86]) to reach consensus, regardless of the severity of failures (stopping, omission, or Byzantine). In [DLS88], the problem was studied using a model of partial synchrony in which upper bounds on message delivery time and/or processes' relative step rates exist, but they are unknown a priori to the processes. The algorithms of [DLS88] are concerned with fault tolerance rather than timing efficiency, and therefore translate to relatively slow algorithms for our model.

For our semi-synchronous model, a lower bound of $(f+1)d$ is implied by the synchronous lower bound of $f+1$ rounds, via a straightforward transformation of any algorithm for our model to an an algorithm for the synchronous model. For stopping and omission failures, any synchronous round-based algorithm may be simulated directly, yielding an algorithm for our model with a running time approximately C times the synchronous running time. This simulation strategy is described in Section 3.1. Thus, upper bounds of approximately $(f+1)Cd$ are easily derived. For Byzantine failures, it is not clear how to simulate a synchronous algorithm correctly.

In [ADLS90], Attiya, Dwork, Lynch, and Stockmeyer prove nearly tight upper and lower bounds on the time to reach consensus in the presence of stopping failures. Surprisingly, they give a clever algorithm for consensus that runs in time $2fd + Cd$, much faster than a direct simulation when C is large. They also show a lower bound of $(f-1)d + Cd$ in a proof that combines the arguments of the synchronous lower bound with techniques from asynchronous lower bounds and retiming techniques for our semi-synchronous model.

1.2 Related work

Current research also concentrating on the real time complexity of the consensus problem appears in [SDC90]. There, processes are assumed to have clocks that are synchronized to within a fixed additive error. In contrast to our results, the results of [SDC90] are stated in terms of process clock time, not absolute time. The relationship between those results and ours is unclear; a better understanding of the differences between two different models is posed as a direction for further research in Section 6.2.

A related model is studied in [HK89] to explore the time complexity of detecting failures along a network path. This model assumes synchronous processes but differentiates between the (known) a priori worst-case bound on message delay, Δ, and the (unknown) actual worst-case message delay in a given execution, δ. Since δ may be much less than Δ, it is desirable for algorithms to have minimal dependency on Δ. This model raises a concern similar to that raised by our model: detecting the absence of a message may be much more costly than receiving the message. Our algorithms run equally well in this model; we remark on how our bounds translate to this model in Section 6.1.

Other work in this area includes the extensive literature on clock synchronization algorithms (see [SWL86] for a survey). Other problems recently studied in our model of timing uncertainty include the problem of mutual exclusion ([AL89]) and the complexity of a network synchronizer algorithm ([AM90]).

1.3 Results of this thesis

1.3.1 Consensus in the presence of omission failures

In Chapter 3, we strengthen the algorithm of [ADLS90] to tolerate omission failures. The resulting algorithm has a running time of $4(f+1)d+Cd$ for $n \geq 2f+1$. This is approximately within a constant factor (4) of the lower bounds of $(f+1)d$ and $(f-1)d+Cd$ ([ADLS90]) and minimizes the dependence on the timing uncertainty C.

For $n \leq 2f$, a more involved analysis bounds the running time by two different quantities simultaneously: one bound is dependent on the ratio $\frac{f}{n-f}$ and the other is dependent on $\sqrt{C}$. We first derive the bound $(3\frac{f}{n-f}+5)(f+1)d+Cd$ using a finer analysis that is similar in spirit to the analysis for $n \geq 2f+1$. We then show that $(2\sqrt{C}+6)(f+1)d+Cd$ is also a bound on the running time using a simple but different argument.

1.3.2 Consensus in the presence of Byzantine failures

In Chapter 4, we present a simulation algorithm using $3f+1$ processes and tolerating f arbitrary failures. The algorithm simulates any synchronous round-based algorithm tolerant of f arbitrary failures using roughly time $2Cd+d$ per round.

The simulation works by keeping processes loosely synchronized to ensure that a nonfaulty process does not advance to round r until it has received a round $r-1$ message from every nonfaulty process. The partial synchronization works by using a combination of two criteria for advancing to further phases, one based on elapsed local time and the other based on messages received.

It follows that any of the known synchronous consensus algorithms tolerating f Byzantine failures and taking $f+1$ rounds can be run in our model in time $(f+1)(2Cd+d)$.

1.3.3 Timeouts using bounded-capacity message links

In Chapter 5, we define a realistic restriction on the message links of our model and examine its effect on the time needed to detect stopping failures. According to the model of [AL89] and [ADLS90] (used in Chapter 3), every message sent by a process is delivered within time d of when it is sent, regardless of the rate at which messages are sent. In reality, if a link is flooded with messages, their delay may be much greater. Our algorithm for omission failures and the algorithm of [ADLS90] ignore this consideration by requiring a process to send a message at every step it takes. This enables failures to be detected as quickly as possible, but is grossly inefficient in its use of messages. We therefore define a more realistic model of message delay that takes into consideration the rate at which messages are sent.

We give a clean, modular definition of a message link of arbitrary capacity μ. Such a link my may be thought of as allowing the "progress" of only μ messages at any time. We then derive nearly tight bounds on the time needed to detect a stopping failure using such links. Two easy algorithms guarantee that the time between a failure and its detection is at most $2Cd + d$ and $C^2d/\mu + Cd + d$, respectively. We show that these bounds are nearly optimal by proving a lower bound of the lesser of $2Cd + d/\mu$ and $C^2d/\mu + Cd + d$.

Chapter 2

Model and Definitions

Our underlying formal model is essentially the same as that used in [ADLS90]. Our model differs by assuming for ease of presentation that all messages are delivered in the order sent and that processes begin executing the algorithm at the same time. The former assumption is not used in our algorithm for Byzantine failures and is easily removed from our algorithm for omission failures by employing a more complicated protocol for receiving messages. The latter assumption is avoided in [ADLS90] by instead providing a special individual *input* event for each process, in which it receives its initial value for the consensus protocol. In measuring the time complexity of the algorithm, time is measured only from the earliest time that all processes have received an input. Using the same formalism, our algorithm for omissions failures works equally well without the assumption of a synchronized start. This is not true, however, for our algorithm tolerating Byzantine failures, where we make use of the fact that all nonfaulty processes begin executing the algorithm at the same time. Without this assumption, the problem is complicated by the need to determine when all processes have received inputs. Also, in addition to allowing stronger failures than [ADLS90], we assume that processes know the number of failures, f, to be tolerated.

2.1 Formal model

We consider a system of n processes $1, \ldots, n$. Each process is a deterministic state machine with possibly an infinite number of states and a distinguished start state.

A *configuration* is a vector C consisting of the local states of each process. Let $st(i, C)$ denote the state of process i in configuration C. We model a computation of the algorithm as a sequence of configurations alternated with *events*. Each event π is either the computation step of a single process or the delivery of a message to a process. The local protocol of process

i consists of two transition functions, M_i for message delivery events, and S_i for computation events. Transition function M_i is applied to a state of the process *and* a message (taken from some finite message alphabet) and returns a state. (So, for example, a process can "remember" a message that was delivered to it.) A message delivery event π is of the form (m, i), specifying the message m delivered and the recipient process, i. Transition function S_i is a applied to a state of the process and returns a state and a finite set of messages to be sent.[1] A computation step π is of the form (i, M), specifing the process i taking the step and the set of messages M it sends in that step. (M should be interpreted as the messages the process *actually* sends at that step in the execution; if the process is faulty, this may not correspond to those determined by the transition function.)

An *execution* is an infinite sequence of alternating configurations and events, $\alpha = C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots$, where C_0 is the vector of start states and each configuration C_i follows from the previous configuration C_{i-1} and the intervening event π_i, according to the state transitions of the process at which event π_i occurs. This means that if event π_j is an event at process x then (1) for $y \neq x$, $st(y, C_{j-1}) = st(y, C_j)$, (2) if π is a message delivery event specifying the delivery of message m then $st(i, C_j)$ is the result of applying M_i to C_{j-1} and m, and (3) if π is a computation event, then $st(i, C_j)$ is the result of applying S_i to C_{j-1}. Also, each message sent is delivered after it is sent and no unsent "messages" are delivered.

A *timed event* is a pair (π, t), where π is an event and t, the "time", is a nonnegative real number. A *timed sequence* is an infinite sequence of alternating configurations and timed events $\alpha = C_0, (\pi_1, t_1), C_1, \ldots, (\pi_j, t_j), C_j, \ldots$, where the times are nondecreasing and unbounded.

Fix real numbers c_1, c_2, and d, where $0 < c_1 \leq c_2 < \infty$ and $0 < d < \infty$. Letting α be a timed sequence as above, we say that α is a *timed execution* if

1. $C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots$ is an execution;
2. The first step of each process is at time 0;
3. There are infinitely many computation steps for each process;
4. If π_i and π_j are consecutive computation steps of the same process, then $c_1 \leq t_j - t_i \leq c_2$; and
5. If message m is sent to process i during computation event π_j then it is delivered to process i during message delivery event π_k, $j < k$, such that $0 \leq t_k - t_j \leq d$.

In our timing analysis (but not in our algorithms or correctness proofs), we make the assumption that $c_2 \ll d$ and therefore make the approximation $d + c_2 \approx d$.

[1] In all our algorithms, a process always sends the same message (at most one per step) to all processes, including itself.

2.1.1 Omission failures

A process i suffers an *omission failure* in execution α if and only if there is a computation step π_j of process i in α specifying a set of messages that is a strict subset of the messages determined by the transition function S_i applied to $st(i, C_{j-1})$. Recall that computation step π_j specifies the messages actually sent by i during that step of execution α. Note that according to our definition of an execution, $st(i, C_j)$ must be the result of applying S_i to $st(i, C_{j-1})$, regardless of the messages specified by π_j. This implies that the process itself is "unaware" of its failure and, unless informed about it, continues executing as if it had not failed. (This kind of failure is sometimes called a *send-omission failure.*) If the algorithm requires j to broadcast a message to all processes, but j does not send a message to i, then we say that "j omits to i" or that this broadcast is "unsuccessful".

2.1.2 Byzantine failures

A process suffers a *Byzantine failure* if it changes its state or sends messages in a way not specified by the transition functions of the algorithm. No restrictions are made on its state transitions or what messages it sends, and so it may exhibit arbitrary behavior. Furthermore, the time between successive steps of a faulty process might *not* be in the interval $[c_1, c_2]$. The messages it sends, however, are delivered within time d of when they are sent.

2.1.3 Consensus

Finally, we define the consensus problem. We assume that each process begins with an initial binary value (its "input") as part of its local state and may irreversibly "decide" on a value by entering a specially designated state. The problem is for the processes to agree on a binary value despite the failure of some processes. We say that a timed execution α is *f-admissible* if at most f processes fail in α. An algorithm *solves the consensus problem for f failures within time T* provided that for each of its *f-admissible* timed executions α, (1) no two different processes decide on different values (agreement), (2) if some nonfaulty process decides on v, then some process has initial value v (validity), and (3) every nonfaulty process decides by time T (time bound). Note that the validity condition does not imply termination; termination is implied by the third condition. We consider the binary version of the problem, where the initial values are 0 or 1. Like the algorithm of [ADLS90], our algorithm for omission failures can be extended to work for any value set, using the same extension given there ([ADLS90], Section 5.4). Our algorithm for Byzantine failures is a general simulation for any rounds based algorithm and therefore can simulate any synchronous agreement algorithm for any value set.

Chapter 3

Consensus in the Presence of Omission Failures

In this chapter, we present a consensus algorithm tolerant of send-omission failures. The algorithm uses the same strategy as that of [ADLS90]; we first elucidate this strategy by describing a *synchronous* consensus algorithm upon which it is based and explaining our algorithm in terms of that synchronous algorithm. For $n \geq 2f+1$, the running time of our algorithm is $4(f+1)d + Cd$, which is approximately within a factor of 4 of the lower bounds of $(t-1)d + Cd$ and $(t+1)d$ ([ADLS90]). For $n \leq 2f$, the running time is bounded by two quantities, $(3\frac{f}{n-f} + 5)(f+1)d + Cd$ and $(2\sqrt{C} + 6)(f+1)d + Cd$.

In order to motivate the work presented here, we first discuss bounds attainable by more straightforward algorithms.

3.1 Straightforward upper bounds

Attiya, Dwork, Lynch, and Stockmeyer ([ADLS90]) give two simple algorithms tolerant of stopping failures and with running times of roughly fCd. One algorithm is based on a method for simulating any synchronous round-based algorithm; the other is specific to the consensus problem and requires that the processes begin synchronized. Both algorithms can be modified to tolerate omission failures without seriously affecting the running times. We briefly explain these two simple algorithms with the modifications.

The first simple algorithm simulates any synchronous round-based algorithm and takes at most time $Cd + d$ per round. The algorithm works by executing the round-based algorithm in parallel with a timeout task. The timeout task is similar to the one described at the beginning of Chapter 5: each process keeps a count of the number of steps it has taken and at each step broadcasts the number of its current step to all other processes in the form "I'm alive: s" at step number s. Each process also keeps track of the "I'm alive" messages received from other processes and detects failures in the expected way, by detecting gaps in the step numbering or by the absence of messages. (We will in fact employ this strategy in our algorithm.) While performing the timeout task, a process simulates each round of the synchronous algorithm by asynchronously executing it—a process simply waits indefinitely on every other process for either a message of that round or the detection of that process's failure. It is not hard to see that this accurately simulates the round-based algorithm: no process sends a round r message before receiving a round $r-1$ message from all nonfaulty processes; A simple inductive argument shows that by time $r(Cd+d)$ (more accurately, time $C(d+c_2)+(d+c_2)$), every process has finished simulating round r of the synchronous algorithm. Thus, any synchronous consensus algorithm tolerant of omission failures taking $f+1$ rounds may be directly simulated to yield an algorithm for our semi-synchronous model that takes time $(f+1)(Cd+d)$.

Under the assumption that processes begin executing the algorithm at the same time, a simpler algorithm specific to the consensus problem may be used. This simpler algorithm does not make use of any fault-detection mechanisms. If a process starts with initial value 1, it broadcasts a 1 and decides 1 and halts. If a process ever receives a 1 (and has not yet halted), it does the same. It is easy to see that if a correct process receives a 1, then some correct process receives a 1 by time fd and subsequently all correct processes receive a 1 by time $(f+1)d$ (more accurately, $(f+1)(d+c_2)$). Therefore a process may decide 0 if it has run for more than $(f+1)(d+c_2)/c_1$ steps without deciding. This takes at most time approximately $(f+1)Cd$.

Finally, we remark that the efficient algorithm of [ADLS90] can be modified to tolerate omission failures by using the timeout task for omission failures outlined above. The running time, however, is then roughly f^2d+Cd. This bound follows from a modification of the part of the analysis of [ADLS90] which takes the sum over each phase r of the number of processes that fail during the sending of an r message. Because only stopping failures are considered in [ADLS90], the analysis there concludes that a process may fail during the sending of at most one r message and therefore the sum over all r is at most f. If failures are by omission, then a process may fail during the sending of many r messages, but only once for any r. Because there are at most $f+2$ phases in any f-admissible execution, the sum over all r is at most $(f+1)f$, resulting in a bound of approximately $(f+1)fd+Cd$.

3.2 Intuition: the underlying synchronous algorithm

Our algorithm and the algorithm of [ADLS90] may be interpreted as simulations of an underlying synchronous algorithm. In this underlying synchronous algorithm, all processes begin executing in round 0. In even numbered rounds, processes may decide only on 0; in odd numbered rounds, processes may decide only on 1. In round 0, any process with initial value 0 decides 0 immediately and broadcasts a message saying "I decided in round 0"; any process with initial value 1 broadcasts a message saying "I didn't decide in round 0" and advances to round 1. In any subsequent round r, if a process did not receive a message saying "I decided in round $r-1$", it may decide $r \bmod 2$, broadcasting "I decided in round r"; if it did receive a message saying "I decided in round $r-1$", it advances to round $r+1$ broadcasting "I didn't decide in round r".

It is easy to see that if a nonfaulty process decides in round r then no process decides in round $r+1$ and all processes then decide in round $r+2$. The algorithm is also "early-stopping": any execution in which at most f processes fail takes at most $f+2$ rounds of communication. (This means that all processes decide in round $f+2$ or earlier, despite the fact that the first round is numbered 0, since a decision in round i is based on messages sent in round $i-1$ or earlier.) The is easily seen by observing that if an execution takes x rounds then a faulty process decides in each of rounds 0 through $x-3$: if no faulty process decides in round $i \leq x-3$ then either (1) a nonfaulty process decides in round i and all processes decide by round $i+2$, or (2) no process decides in round i and therefore they all decide in round $i+1$ (because no process receives an "I decided in round i" message). Thus, f failures cause the maximum number of rounds, $f+2$, in the following execution. All processes except some process j_0 begin with initial value 1 and advance to round 1. Process j_0, with initial value 0, broadcasts "I decided in round 0" to all processes except some other process j_1. Thus all processes except j_1 advance to round 2; j_1 decides in round 1 and broadcasts "I decided in round 1" to all processes except some process j_2. This continues until finally process j_{f-1} decides in round $f-1$ and broadcasts "I decided in round $f-1$" to all processes except nonfaulty process j_f, which decides in round $f+1$; all processes subsequently decide in round $f+2$.

Both our algorithm and that of [ADLS90] "simulate" this synchronous algorithm, making several important optimizations in order to improve the running time for our model. If during the simulation of round r, a process receives a message saying "I decided in round $r-1$", it immediately advances to round $r+1$ (without waiting for round $r-1$ messages from other processes), broadcasting to all processes, in effect, "I *know of a process that* decided in round $r-1$". Other processes in round r that receive this message relay it to all processes and also advance immediately to round $r+1$. A process may decide in round r only if it can be sure that no nonfaulty process decided in round $r-1$. This is ascertained only when, for every other process p, either (1) the message "I didn't decide in round $r-1$" is received from p, or (2) p has been detected as faulty (by the timeout protocol), or (3) for some $r' < r-1$, the message "I decided in round r'" has been received from p (also remembered by the timeout protocol).

The key to the improved efficiency of our algorithm relative to that of [ADLS90] is the addition of a mechanism for a process to detect *its own* failure. We require that a process receive at least $n-f$ acknowledgments for every message of the synchronous algorithm that it sends. Until a process has received a sufficient number of acknowledgments for its round r message, it is prohibited from deciding in round $r+1$ or advancing to round $r+2$. This is important to the efficiency of the algorithm because it limits to 1 the number of times a faulty process can omit a message of the synchronous algorithm to all nonfaulty processes. For $n \geq 2f+1$, the convention of waiting for acknowledgments ensures that a faulty process does not advance to round $r+1$ if it omits to all nonfaulty processes a message saying "I know of a process that decided in phase r". If it does send such a message to a nonfaulty process, that nonfaulty process in turn relays it to all other processes; the faulty process therefore has not delayed the algorithm by very much (time d at most). The convention of waiting for acknowledgments requires that a process continue executing the algorithm, sending acknowledgments, after it has decided.

3.3 The algorithm

We first explain the presentation of our algorithm. We describe our algorithm as the parallel composition of a fault-detection protocol and a main algorithm. At each step, a process first executes the code of the fault-detection protocol, then executes the code of the main algorithm, and finally sends a message. (Recall that in our model a process may send at most one message at each step).

This message is the concatenation of possibly several component "messages" which are specified by the **queue** commands in the code: if during a step, the statement "**queue** 'm'" is executed in the code, then "message" m is a component of the message sent at the end of that step. We will refer to a message by any one of its components: we will say "an m message" or simply "an m" to refer to any message with m as one of its components.

Our model also specifies that a process receives messages only during delivery events (and therefore only between process steps). For every delivery event, a process changes its state by adding the received message to a buffer (an unordered set). At its next step, the process reads and empties this buffer. A conditional statement in the code referring to the receipt of a message checks whether such a message was read from this buffer during the given step.

For ease of presentation, some components of a process's state are not explicitly named or maintained in the code—for instance, the number of steps a process has taken, whether it has decided, or whether it has sent a certain message. Process index subscripts are omitted in the code but used in the text (e.g., "D_i") to refer to a local variable (D) of process i.

3.3.1 The fault-detection protocol

In order to tolerate omission failures, our algorithm employs the timeout protocol described in Section 3.1. A process sends a message at every step that it takes, consecutively numbering all messages that it sends with the number s of its current step.[1] Before a process decides, the message that it sends at every step is of the form "I'm alive: s", where s is the number of its current step; after a process decides, the message is of the form "I've decided: s". The failure of a process can thus be detected by a gap in the sequence numbering (recall we assume that message links deliver messages in the order sent) or by the absence of any messages for too long a period of time (more than time $d + c_2$).

All processes detected as faulty are added to a local set F. When a process i detects the failure of another process j, it broadcasts this fact in the form of a "shutdown j" message. Upon receiving this message, other processes add j to their respective sets F; when process j receives this message, it *halts*, ceasing its execution of the algorithm. The timeout protocol also keeps track of which processes have decided. When a process receives a message "I've decided: s" from another process, it adds that process to its set D. When a process i adds j to D_i (F_i, resp.), it is said to have "detected" that j has decided (failed, resp.). We say that a process i is *shut down at time t* if it receives a "shutdown i" message at time t. The code for the fault-detection protocol is in Figure 3.1.

We now verify two basic properties of the fault-detection protocol with respect to arbitrary executions. The first bounds the time by which a failure is detected.

Lemma 3.1 *If at time t, process j omits a message to process i, and i is not shut down by time $t + C(d + c_2) + (d + c_2) \approx t + Cd + d$, then i adds j to F_i by that time.*

Proof: Let s_j be the step number of j at which it omits a message to i. The lemma is clearly true if j sends a message to i at a step numbered greater than s_j and that message arrives at i by time $t + C(d + c_2) + (d + c_2)$. If j does not send such a message, then i receives no message from j between time $t + d$ and $t + d + c_2(1 + (d + c_2)/c_1) = t + (d + c_2) + C(d + c_2)$, in which time i takes more than $(d + c_2)/c_1$ steps and, since it is not yet shut down, adds j to F_i. ■

[1] As a consequence of the bound on running time to be derived, these sequence numbers are bounded by a function of f, d, c_1 and c_2.

STEP s: **If** "shutdown i" received, **then halt.**
If decided, **then queue** "I've decided: s"
else queue "I'm alive: s".
For each $j \notin D \cup F$,
if "shutdown j" message received
then $F \leftarrow F \cup \{j\}$; **queue** "shutdown j"
if "I'm decided: s_j" message received from j
then $D \leftarrow D \cup \{j\}$
if "I'm alive" messages from j not numbered consecutively
then $F \leftarrow F \cup \{j\}$; **queue** "shutdown j"
if no message received from j and more than
$(d + c_2)/c_1$ steps taken since last message received from j,
then $F \leftarrow F \cup \{j\}$; **queue** "shutdown j".

Figure 3.1: The fault-detection protocol for i at step number s.

The second property verifies that nonfaulty processes are never declared faulty.

Lemma 3.2 *If process i does not fail in an execution, then i is not added to any set F_j and is never shut down.*

Proof: For contradiction, let j be the process that first adds i to its failed set F_j. Process j adds i to F_j because either it receives a "shutdown i" message, or it receives two "I'm alive" messages from i with a gap in sequence numbering, or it does not receive an "I'm alive" message from i for more than $(d + c_2)/c_1$ steps.

By our choice of j, process j cannot receive a "shutdown i" message before adding i to F_j—that would imply that some other process added i its failed set before j did.

Because i is nonfaulty (and the links are FIFO), j does not receive two "I'm alive" messages with a gap in the sequence numbering.

Before it decides, i sends "I'm alive" messages at every step it takes and so any two messages are delivered to j at most time $d+c_2$ apart (if one message is delivered immediately and the following message is delayed by d). In time $d + c_2$, j can take at most $(d + c_2)/c_1$ steps and therefore does not add i to F_j. After i decides, it broadcasts an "I'm decided" message, which causes j to add i to D_j and prevents j from adding i to F_j thereafter. Thus, j cannot add i to F_j. ∎

3.3.2 The main algorithm

The main algorithm is basically an asynchronous version of the synchronous algorithm of Section 3.2. The code for the main algorithm appears in Figure 3.2. We call the simulation of round r of the synchronous algorithm "phase r". Each process i starts in phase 0 with v_i set to its own private value (1 or 0). In its first step, a process either decides 0 or advances to phase 1. As with the synchronous algorithm, in even numbered phases a process can decide only 0, and in odd numbered phases a process can decide only 1.

When a process advances from phase r to phase $r+1$, it broadcasts an "r" message. (This is the equivalent of the message "I didn't decide in round r" in the synchronous algorithm). When a process decides in phase r, it broadcasts an "$r+1$" message. (The message "$r+1$" replaces the messages "I decided in round r" and "I didn't decide in round $r+1$" of the synchronous algorithm; both have the meaning "I know of a process that decided in round r", so it is unnecessary to distinguish between them.) Set M^r contains those processes from which an r message has been received. A process may decide in phase r only if it has (1) not yet received an r message, and therefore does not know of a process that decided in round r, and (2) has received an $r-1$ message from all processes not yet detected as faulty or decided, indicating that they did not decide in round $r-1$. If process i is nonfaulty, then the receipt of an $r+1$ message from i prevents other processes from deciding in phase $r+1$ since they do not add i to D or F before receiving it. A process that decides in round r does not send an r message unless it receives one first (this implies that some process decided in round $r-1$ but failed).

Our convention of acknowledging messages works as follows. Each process maintains a set A^r containing those processes from which a properly sequenced "ack$(\cdot, r)$" message has been received. (The restriction to properly sequenced "ack$(\cdot, r)$" messages is achieved by not adding a process to A^r if that process is already in F. This restriction is necessary only for the bound when $n \leq 2f$.) Until a process decides, it sends exactly one acknowledgment message, "ack(j, r')", for each r' message where r' is less than its current phase number. After a process decides in some phase r, it continues to acknowledge r' messages for $r' \leq r+1$. This is implemented in the code by allowing the process to advance to phase $r+2$ but no further. It is not necessary for a process to acknowledge r' messages for $r' > r+1$ because as we will see, if it is nonfaulty then other nonfaulty processes do not advance to phase $r+3$ without deciding and therefore do not require acknowledgments for their $r+2$ messages. Until a process has received at least $n-f$ properly sequenced acknowledgments for its r message ($|A^{r-1}| \geq n-f$), it may not advance to phase $r+1$ or decide in phase r.

Definition 1 *A process i is* blocked in phase r *(for $r > 0$) if it advances to phase r without deciding and never has $|A_i^{r-1}| \geq n-f$.*

Being blocked is a permanent state, but even if a process is not blocked in phase r, it may be temporarily *delayed* from advancing to phase $r+1$ as it waits for acknowledgments before proceeding.

PHASE 0: **If** $v = 1$, **then queue** "0" and **goto** PHASE 1.
If $v = 0$, **then queue** "1" and **decide** 0 and **goto** PHASE 2.

PHASE $r > 0$: **For each** j and **each** r', $1 \le j \le n$ and $0 \le r' < r$,
if "r'" message received from j,
then $M^{r'} \leftarrow M^{r'} \cup \{j\}$
if "ack$(i, r-1)$" received from j and $j \notin F$,
then $A^{r-1} \leftarrow A^{r-1} \cup \{j\}$
if $j \in M^{r'}$ and $r' < r$ and "ack(j, r')" not yet sent,
then queue "ack(j, r')". (whether decided or not)
If decided and $M^{r-2} \neq \emptyset$ and "$r-2$" not yet sent,
then queue "$r-2$"
If not decided and $|A^{r-1}| \ge n - f$, (enough ack's received)
then if $M^r \neq \emptyset$, (some process decided in phase $r-1$)
then queue "r" and **goto** PHASE $r+1$
if $M^r = \emptyset$ and $j \in M^{r-1}$ for all $j \notin (D \cup F)$,
then queue "$r+1$" and **decide** r mod 2 and **goto** PHASE $r+2$

Figure 3.2: The main algorithm of process i, performed at every step. Initially, a process is in phase 0 with $M^r = A^r = \emptyset$ for all r.

We prove here a few basic lemmas about the main algorithm with respect to any f-admissible execution. The first two lemmas affirm two expected properties that held for the synchronous algorithm.

Lemma 3.3 *If some nonfaulty process decides in phase $r \ge 0$ then no process decides in phase $r+1$.*

Proof: Let i be a nonfaulty process that decides in phase r and consider any other process j. According to the code of the main algorithm, j cannot decide in phase $r+1$ without receiving an r message from i or adding i to F_j or D_j. We claim that neither can happen before j receives an $r+1$ message from i, which according to the code (which requires $M^{r+1} \neq \emptyset$) precludes j from deciding phase $r+1$. First, because i is nonfaulty, by Lemma 3.2 it is never added to F_j. Process i may send an r message, but only after sending an $r+1$ message. Because i is nonfaulty, it does not omit this message, and because messages are delivered in the order sent, j does not receive it before receiving the $r+1$ message. Process i is added to D_j only when j receives the message "I've decided" from i. By the same argument, j does not receive "I've decided" from i before receiving the $r+1$ message. ■

The following definition is useful in proving correctness and analyzing time complexity.

Definition 2 *Phase r is* quiet *if there is some process that never receives any r messages.*

Lemma 3.4 *If a nonfaulty process decides in phase $r \geq 0$ then phase $r+2$ is quiet.*

Proof: By Lemma 3.3, no process decides in phase $r+1$. If a process does not decide in phase $r+1$, then it does not send an $r+2$ message until it receives one. Therefore, no process sends an $r+2$ message and in fact no process receives an r message. ■

The next two lemmas affirm that the convention of acknowledging r messages works as expected—nonfaulty processes are never blocked—and the last lemma states that the failure of blocked processes is eventually detected by all processes.

Lemma 3.5 *For any process i and any nonfaulty process j, if i advances to phase $r \geq 1$ without deciding and sends an r' message j for $0 \leq r' \leq r-1$, then i receives an "ack$(i, r-1)$" message from j.*

Proof: By induction on r. Clearly the lemma is true for $r = 1$: j advances to phase 1 during its first step and sends "ack$(i, 0)$" during the next step at which it has received a 0 message from i.

Assume the lemma is true for $r-1 \geq 1$. First observe that j does not decide in any phase $r' \leq r-3$: by Lemma 3.3, this would imply that no process decides in phase $r'+1$ and therefore no process sends an $r'+2$ message, but this is not possible because i advances to phase $r \geq r'+3$ without deciding and therefore must receive an $r'+2$ message. If j decides in phase r' and $r' = r-2$ or $r-1$, then j immediately advances to phase $r'+2 \geq r$ after deciding and sends "ack$(i, r-1)$" to i. Suppose that j does not decide in any phase $r' \leq k-1$. Process j must advance from each phase $r' \leq k-1$ because it is never shut down, has $M_j^{r'} \neq 0$, and has $|A_j^{r'}| \geq n-f$: j is never shut down by Lemma 3.2; j has $M_j^{r'} \neq \emptyset$ because it receives an r' message from i; j has $|A_j^{r'}| \geq n-f$ because it is nonfaulty and therefore sends an r'' message to all processes for each $r'' \leq r'-1$ and by the induction hypothesis receives "ack(j, r'')" from all nonfaulty processes—none of which, by Lemma 3.2 are ever added to F_j. Process j therefore advances to phase r and may then send "ack$(i, r-1)$" to i. ■

Corollary 3.6 *If process i is nonfaulty and advances to phase $r \geq 1$ without deciding, then it eventually has $|A_i^{r-1}| \geq n - f$. (A nonfaulty process is never blocked.)*

Proof: Because i is nonfaulty and advances to phase r without deciding, for $0 \leq r' < r$ it sends an r' message to all processes as it advances to phase $r' + 1$. By Lemma 3.5, i receives "ack$(i, r-1)$" from each nonfaulty process. Because by Lemma 3.2, nonfaulty processes are never added to F_i, each nonfaulty process is added to A_i^{r-1}, giving the necessary bound. ∎

The following lemma relies on the fact that a process continues to take steps, executing the algorithm after it decides; in particular, it continues to detect the failure of processes and, if necessary, send acknowledgments.

Lemma 3.7 *If a faulty process j unsuccessfully broadcasts an r message at time t and is subsequently blocked in phase $r+1$, then all processes not shut down by time $t + C(d + c_2) + 2(d + c_2) \approx t + Cd + 2d$ detect the failure of j by that time.*

Proof: By the definition of being blocked, j advances to phase $r+1$ but never has $|A_j^r| \geq n - f$. Thus there is some nonfaulty process i never added to A_j^r. By Lemma 3.5, j omits an r' message to i for some $0 \leq r' \leq r$. This omission occurs at or before time t. By Lemma 3.1, i detects this failure by time $t + C(d + c_2) + (d + c_2)$, broadcasting "shutdown j" to all processes in the same step. By time $d + c_2$ later, all processes not yet shut down have received this message and taken a step, adding j to their failed sets. ∎

3.4 Correctness proof

We now prove that in all f-admissible executions, the algorithm terminates and correctly satisfies the agreement and validity conditions. We first prove "progress"—that processes in fact advance to successive phases as expected. Given this progress lemma and a few simple facts about quiet phases, the proofs of agreement, validity, and termination are easily derivable. These proofs follow the same reasoning as the informal argument about the synchronous algorithm outlined in Section 3.2.

Lemma 3.8 *For each $r \geq 0$ and each process i that is neither blocked nor shut down in any phase $r' \leq r$, process i either decides in some phase $r' \leq r$ or advances to phase $r+1$.*

Proof: For contradiction, let phase r be the first phase for which the lemma is not satisfied and let i be any process for which the lemma is not satisfied at phase r. By the choice of r, i advances to phase r.

First note that $r \neq 0$, since every process either decides or advances to phase 1 during its first step.

We show below that for $r > 0$ and for every process j, either i either receives an $r-1$ message from j or adds j to F_i or D_i. We thus derive a contradiction by concluding that i may either decide or advance to phase $r+1$, since it has $j \in M^{r-1}$ for all $j \notin (D_i \cup F_i)$ and by assumption is not shut down and eventually has $|A_i^{r-1}| \geq n-f$ (is not blocked in phase r).

Let j be any other process. First consider the case that j also is neither shut down nor blocked in any phase $r' \leq r$ and that further, j does not fail directly to i. By the choice of r, j either advances to phase r or decides in a previous phase. If j advances to phase r, then it must send an $r-1$ message to i (successfully, in this case). It cannot be that process j decides in phase $r-1$, since that would imply sending an r message to i, thus enabling i to advance immediately to phase $r+1$, contradicting our original assumption. If j decides before phase $r-1$, then it sends an "I've decided" message to i and is added to D_i.

Now consider the case that j is either shut down or blocked in some phase $r' \leq r$ or j fails directly to i. If j is blocked, then by Lemma 3.7, i will eventually detect that j is faulty. Similarly, if j is shut down, then it halts and i will detect its failure by timeout. Lastly, Lemma 3.1 ensures that if j fails directly to i and i is not shut down, then i eventually detects j as faulty and adds it to F_i. ■

Corollary 3.9 *For any $r \geq 0$, every nonfaulty process either decides in phase $r' \leq r$ or advances to phase $r+1$.*

Proof: By Lemmas 3.2 and 3.6, a nonfaulty process is never shut down or blocked; the corollary then follows immediately from Lemma 3.8. ■

Corollary 3.10 *If phase $r \geq 0$ is quiet, then each nonfaulty process decides in some phase $r' \leq r$.*

Proof: By Corollary 3.9, each nonfaulty process either decides in phase $r' \leq r$ or advances to phase $r+1$. But a nonfaulty process cannot advance to phase $r+1$: to do so, it would send an r message to all processes, contradicting the assumption that phase r is quiet. ■

Lemma 3.11 (Agreement) *No two nonfaulty processes decide on different values.*

Proof: Let r be the first phase in which some nonfaulty process i decides. By Lemma 3.3, no process decides in phase $r+1$. Because no process decides in phase $r+1$, no process sends an $r+2$ message and thus phase $r+2$ is quiet. Thus by Lemma 3.10, all nonfaulty processes decide in some phase $r' \leq r+2$. By the choice of r, all nonfaulty processes decide in either phase r or phase $r+2$, in either case on $r \bmod 2$. ■

Lemma 3.12 (Validity) *If any process decides on value b, then some process i starts with $v_i = b$.*

Proof: Clearly if some process j decides on 1, it does so in phase $r > 0$ and that process itself must have started with $v_j = 1$ since otherwise it would have decided on 0 during its first step.

If some process j decides on 0, it cannot be that all processes started with $v_i = 1$. For then, no process would decide in phase 0 and no process would send a 1 message. No process would receive a 1 message and therefore no process would advance to phase 2 without deciding and so no process would decide 0. ■

Lemma 3.13 (Termination) *In any f-admissible execution, there is a quiet phase numbered at most $f + 2$ and so each nonfaulty process decides in some phase $r \leq f + 2$.*

Proof: If some nonfaulty process decides in phase $r \leq f$ then no process decides in phase $r+1$ and no process sends an $r+2$ message. Phase $r+2$ is therefore quiet and by Lemma 3.10 all nonfaulty processes decide by phase $r + 2 \leq f + 2$.

If no nonfaulty process decides in any phase $r \leq f$, then there must be a phase h, $0 \leq h \leq f$, in which no *faulty* process decides, and therefore in which *no* process decides. If a process does not decide in phase h, then it does not send an $h + 1$ message until it receives one. Therefore no process sends an $h + 1$ message—phase $h + 1$ is quiet—and by Lemma 3.10, all nonfaulty processes decide by phase $h + 1 \leq f + 1$. ■

3.5 Analysis of time bounds

We now bound the amount of real time until all nonfaulty processes decide in any f-admissible execution. The analysis in this section is carried out with respect to any given f-admissible execution. Having already proved the correctness of the algorithm, we will hereafter assume $d \gg c_2$ and make approximations appropriately. We first establish the tools for our analysis and then conclude with the nearly optimal bound for $n \geq 2f+1$ (Section 3.5.1) and two bounds for $n \leq 2f$ (Section 3.5.2). We first introduce some notation.

- For $r \geq 0$, let t_r be the earliest time by which all processes not blocked in any phase $r' \leq r$ of the execution have either decided, advanced to phase $r+1$, or been shut down.
  Because every process either decides or advances to phase 1 on its first step, $t_0 = 0$.
- Let phase h be the first (smallest numbered) phase that is quiet.
- For $r \geq 0$, let $B_r = \{i : i \text{ is blocked in phase } r+1\}$; let $b_r = |B_r|$.

The definition of B_r may seem unusual, but makes sense on closer analysis. We will want to bound $t_r - t_{r-1}$, which we think of as the time for phase r, in terms of the number of processes that omit an r message to all nonfaulty processes. This number is b_r, since all such processes are subsequently blocked in phase $r+1$.

Lemma 3.14 *For $r \neq r'$, $B_r \cap B_{r'} = \emptyset$.*

Proof: By definition, a process must advance to phase r' in order to be blocked in phase r'. If $r < r'$ and $i \in B_r$, then i is blocked in phase $r+1 \leq r'$ and cannot advance to phase $r+2 \leq r'+1$ or greater. Therefore, i is not blocked in phase $r'+1$ and cannot be in $B_{r'}$. ■

Corollary 3.15 $\sum_{i=0}^{i=\infty} b_i \leq f$.

Proof: By Corollary 3.6, a nonfaulty process is not in any B_r, so these sets consist of faulty processes only. The bound of f then follows immediately from the disjointness of the sets B_r, from Lemma 3.14. ■

We prove our upper bound by summing the times of the individual phases. We will say "the time of/for phase r" to mean $t_r - t_{r-1}$. We prove an upper bound for two kinds of phases: those that are quiet and those that are not. We first derive some useful lemmas about the receipt of acknowledgments. We then prove an upper bound on the time to complete any phase—in particular, quiet phases. We then prove a lemma (Lemma 3.19) that is at the heart of the timing analysis, regarding causal chains of r messages. The time for phases that are not quiet depends on whether or not $n \geq 2f+1$ and will be deferred until the following subsections (Sections 3.5.1 and 3.5.2), where we will also sum over the phases to derive the total time bounds.

We first prove a useful lemma about the timeliness of acknowledgments: if a process receives a sufficient number of properly sequenced acknowledgments for its r message, then it receives them promptly, by time $t_{r-1}+2d$.

Lemma 3.16 *For $r \geq 0$, if process j eventually has $|A_j^r| \geq n-f$, then it has $|A_j^r| \geq n-f$ by time $t_r + 2d$.*

Proof: Process j sends an r message either as it advances to phase $r+1$ or as it decides in phase $r-1$. If process j broadcasts its r message because it advances to phase $r+1$, then it is clearly not blocked in any phase $r' \leq r$ and is neither decided nor shut down before it broadcasts this message, and so broadcasts it by time t_r. Similarly, if j broadcasts its r message because it decides in phase $r-1$, then it does so by time t_{r-1}. In either case, j broadcasts its r message by time t_r and any process that receives an r message from j receives it by time $t_r + d$.

Consider any process $i \in A_j^r$. We claim that i sends "ack(j,r)" by time t_r+d. By the fact that it sends "ack(j,r)" eventually, process i must advance to phase $r+1$ or greater (either by deciding in phase $r-1$ or phase r or by advancing to phase $r+1$ without deciding) before sending "ack(j,r)". It follows that i is neither blocked in any phase $r' \leq r$ nor shut down before it does so and therefore advances to phase $r+1$ by time t_r. By time t_r+d, i also receives an r message from j and therefore sends "ack(j,r)" by then. ■

Corollary 3.17 *For $r \geq 0$, if process i sends an $r+1$ message after time t_r or for some j process i sends "ack(j,r)", then i has $|A_j^r| \geq n-f$ by time t_r+2d.*

Proof: If process i sends an $r+1$ message after time t_r then it does not send the $r+1$ message as a result of deciding in phase r, since processes that decide in phase r do so by time t_r. Therefore i sends an $r+1$ message as a result of advancing from phase $r+1$, which requires that it have $|A_j^r| \geq n-f$. By Lemma 3.16, i therefore has $|A_j^r| \geq n-f$ by time t_r+2d. ■

We now prove a generous upper bound on the time to complete any phase (in particular, quiet phases). The proof is very similar to the proof of progress.

Lemma 3.18 *$t_1 - t_0 \le Cd + d$ and for any phase $r > 1$,*
$t_r \le \max(t_{r-1} + Cd + d,\ t_{r-2} + Cd + 2d)$.

Proof: For contradiction, assume that for $r > 1$ (respectively, $r = 1$) at time $\max(t_{r-1} + Cd + d,\ t_{r-2} + Cd + 2d)$ (resp. time $t_0 + Cd + d$), some process i has neither decided nor advanced to phase $r + 1$ nor been shut down and is never blocked in any phase $r' \le r$. By this time, by the definition of t_{r-1}, i is in phase r, and since it is not blocked in phase r, has $|A_i^{r-1}| \ge n - f$ by Lemma 3.16. We will reach a contradiction by showing that i must decide in phase r by this time because for every other process j, either i receives an $r - 1$ message from j or i detects that j has decided or failed ($j \in D_i \cup F_i$).

Let j be any other process. First consider the case that j (1) is not blocked in any phase $r' \le r - 1$, (2) is not shut down by time t_{r-1}, and (3) does not fail directly to i before or at time t_{r-1}. By Lemma 3.8, j either advances to phase r or decides in some phase $r' \le r - 1$; by definition it does so by time t_{r-1}. If j advances to phase r, then it sends (successfully, by assumption) an $r - 1$ message to i by time t_{r-1} and i receives this message by time $t_{r-1} + d$. If j decides in phase $r' \le r - 1$, then by time $t_{r'} + d \le t_{r-1} + d$, i receives an "I've decided" message from j and adds j to D_i.

Now consider the case that j either (1) is blocked in some phase $r' \le r - 1$ (2) is shut down by time t_{r-1}, or (3) fails directly to i at or before time t_{r-1}. If j is shut down or fails directly to i at or before time t_{r-1}, then by Lemma 3.7, i detects the failure by time $t_{r-1} + Cd + d$. Case (1) is not possible for $r = 1$, so we are finished for that case. If j is blocked in some phase $r' \le r - 1$, then because it advances to phase r', j neither decides nor is blocked nor shut down in any prior phase. Therefore, by time $t_{r'-1} \le t_{r-2}$, j advances to phase r', broadcasting (unsuccessfully) an $r' - 1$ message. By Lemma 3.7, all processes, including i, detect the failure of j by time $t_{r-2} + Cd + 2d$. ■

In bounding the time of a phase r that is not quiet, we will bound the time until every process receives an r message (which every process does, by the definition of a quiet phase). By that time, every process that is not yet decided or shut down or blocked in any phase $r' \le r$ may advance to phase $r + 1$; thus this is a bound for t_r. In bounding the time until every process receives an r message, the following reasoning is at the heart of the analysis. In order for the first r message to ever be sent, some process must decide in phase $r - 1$, which by definition, it does by time t_{r-1}. An r message sent by any other process i that does not decide in phase $r - 1$ is sent because i received an r message. Thus, a causal chain of r messages may be followed and the first r message received by any process can be traced back to a process that originated it (i_k in the following lemma), sending the "first" r message before t_{r-1}. Because a process broadcasts an r message as soon as it receives one (at its next step, to be precise; also, assuming it has $|A^{r-1}| \ge n - f$, which it does after $t_{r-1} + 2d$ if at all), our time bound for phases that are not quiet is approximately d times the length of the shortest such chain to each process. We now prove a lemma about the existence of such chains and their basic timing properties. This lemma is central to every bound we will prove for the omission failures algorithm.

Lemma 3.19 *If phase r is not quiet, then for every process i_0, there exists a sequence of distinct processes $i_0, i_1, \ldots, i_k$ and messages $m_0, m_1, \ldots, m_k$ with $k \geq 0$ such that*
(1) for $0 < j \leq k$, i_j sends the first r message, m_{j-1}, received by i_{j-1},
(2) exactly one process, i_k, sends an r message by time t_{r-1}, and
(3) for $0 < j \leq k$, process i_j sends an r message (m_{j-1}) by time $t_{r-1} + (k - j + 1)d$.

Proof: Phase r is not quiet, so every process i_j receives an r message; let m_j be the first r message that i_j receives. Define a sequence of processes $i_0, i_1, \ldots$ inductively as follows: if i_j sends an r message by time t_{r-1} then define $k = j$ and let i_j be the last process of the sequence; otherwise, define i_{j+1} to be the process that sends m_j.

We first claim that the resulting does not include repetitions and is therefore finite. This is clear if i_0 sends an r message by time t_{r-1} (then $k = j = 0$). If not, we show that for any $0 \leq j \leq k$, process i_j is distinct from processes $i_0, \ldots, i_{j-1}$. Only i_0 may fail to send an r message. If it does, then clearly it is distinct from the other processes in the sequence; if not, then let m_{-1} be any r message that it sends. If i_j sends an r message by time t_{r-1}, then clearly it is distinct and we are done. If not, then for all i_x, $0 \leq x \leq j$, because i_x sends an r message (m_{x-1}) later than time t_{r-1}, i_x must send it as the result of receiving an r message (by the definition of t_{r-1}, a process that decides in phase $r-1$ broadcasts r by time t_{r-1}). It follows that the sending of m_{x-1} by i_x is preceded by the sending of m_x, the first r message received by i_x. Because a process broadcasts an r message only once, it follows that processes $i_0, \ldots, i_j$ are distinct.

Thus the sequence $i_k, \ldots, i_1, i_0$ forms a chain of processes such that for $0 < j \leq k$, process i_j sends the first r message, m_{j-1}, received by i_{j-1} and k is the only process in the sequence to broadcast an r message before time t_{r-1}. This proves (1) and (2).

It remains to show (3), the timing property. For $0 < j < k$, the fact that i_j sends an r message but does not decide in phase $r-1$ implies that i_j advances to phase r by time t_{r-1}, since it is not blocked in any phase $r' \leq r$ and is not decided or shut down before sending m_{j-1}, which it does after time t_{r-1}. Therefore, by time $t_{r-1} + 2d$, each i_j is in phase r and by Lemma 3.16, has $|A_{i_j}^{r-1}| \geq n - f$. Since i_{k-1} receives m_{k-1} by time $t_{r-1} + d$, it advances to phase $r+1$, sending m_{k-2} by time $t_{r-1} + 2d$. Process i_{k-2} receives this message by time $t_{r-1} + 3d$ and thus advances to phase $r+1$, sending m_{k-3} by time $t_{r-1} + 3d$. Similarly, for $0 \leq j < k$, process i_j receives m_j and sends m_{j-1} by time $t_{r-1} + (1 + k - j)d$. ∎

To complete the lemmas necessary to tightly bound the running time, we need only bound the time for any phase that is not quiet. This bound depends on whether or not $n \geq 2f + 1$.

3.5.1 Bound for $n \geq 2f+1$

We show that the algorithm depends on C only to the extent of an additive factor of Cd. For C large, this algorithm may be far more efficient that a direct rounds simulation. The bound we obtain for $n \geq 2f+1$ is within approximately a factor of 4 of optimal: our bound is $4(f+1)d + Cd$; the lower bound proved in [ADLS90] is $(f-1)d + Cd$.

Having bounded the time for quiet phases in Lemma 3.18, we need only bound the time for any phase that is not quiet. If $n \geq 2f+1$, we can be sure that when a faulty process broadcasts an r message, it *either sends to at least one nonfaulty process or becomes blocked in phase* $r+1$ since $f < n-f$. If it sends to a nonfaulty process, then that process will send an r message to all processes and the phase will end. The number of processes blocked in phase $r+1$ is exactly b_r; our bound for phase r is roughly $b_r \cdot d$. This is the key difference between our algorithm and the algorithm of [ADLS90]: a faulty process may cause delay d only if it sends exclusively to other faulty processes; the convention of requiring acknowledgments ensures that each faulty process can do so only once.

To reinforce the intuition about this bound, we first describe how this bound is realized by a worst-case execution: Process $1 \in B_r$ is the first to send an r message. It decides in phase $r-1$ at time t_{r-1} (no later, by definition of t_{r-1}, since process 1 is not blocked in any phase $r' \leq r-1$) and sends an r message to only process $2 \in B_r$. Process 2 waits until time $t_{r-1} + 2d$ for $|A_2^{r-1}| \geq n-f$ and then, having received an r message from 1, advances to phase $r+1$, sending an r message to only process $3 \in B_r$. The pattern is repeated until process $b_r + 1 \notin B_r$ receives an r message at time $t_{r-1} + (b_r+1)d$. Process $b_r + 1$ advances to phase $r+1$ and omits an r message to exactly one nonfaulty process, i. All nonfaulty process except i receive an r message from $b_r + 1$ at time $t_{r-1} + (b_r+2)d$ and i receives an r message from them at time $t_{r-1} + (b_r+3)d$. By this time, each process has either advanced to phase $r+1$ (as it sent an r message), decided, been shut down, or is blocked in some phase $r' \leq r$. This scenario shows where the extra $3d$ arises: one d is caused by the delay of waiting (by process 2 in this scenario) for acknowledgments from the previous phase, another d is for a faulty process (here, b_r+1) that is not blocked in phase $r+1$ to send an r message to a nonfaulty process, and another d is for the remaining nonfaulty processes (here, i) to receive an r message. (In [ADLS90], only the last extra d is incurred; this leads to the factor of 2 in their bound, instead of 4 in ours.)

Lemma 3.20 *For $n \geq 2f+1$ and $r \geq 1$, if for all $r' \leq r$ phase r' is not quiet, then $t_r - t_{r-1} \leq (3+b_r)d$.*

Proof: We show that by time $t_{r-1} + (3+b_r)d$, all processes receive an r message. Thus, by that time, every process is either decided, shut down, blocked in some phase $r' \leq r$, or may advance to phase $r+1$.

By Lemma 3.19, we know that for every process i_0, there is a sequence of distinct processes, $i_0, i_1, \ldots, i_k$ satisfying the three properties of Lemma 3.19.

Now, if $k \leq b_r + 2$ then i_0 receives m_0 by time $t_{r-1} + (k-1+1)d + d \leq t_{r-1} + (b_r + 2)d + d$. If $k > b_r + 2$, then there is a j such that $k - b_r \leq j \leq k$ and $i_j \notin B_r$. By Lemma 3.19, i_j sends an r message by time $t_r + (k-j+1)d \leq t_{r-1} + (1 + b_r)d$. Because $i_j \notin B_r$, i_j sends an r message to at least $n - f \geq f + 1$ processes, one of which must be nonfaulty. This nonfaulty process, ℓ, receives an r message from i_j by time $t_{r-1} + (2 + b_r)d$.

We now conclude the proof by showing that process ℓ sends an r message, received by all processes, by time $t_{r-1} + (2 + b_r)d$. Because no phase $r' \leq r$ is quiet, it follows from Lemma 3.4 that ℓ does not decide in any phase $r' \leq r - 2$. If ℓ decides in phase $r - 1$, then does so, sending an r message, by time t_{r-1}. If ℓ decides in phase r or advances to phase r without deciding, then it does so by time t_{r-1} and subsequently sends an r message once it receives one and has $|A_\ell^{r-1}| \geq n - f$, which, by Lemma 3.16, it does by time $t_{r-1} + (2 + b_r)d$. ∎

We can now bound tightly the running time of any f-admissible execution by summing the bounds for all phases in that execution.

Theorem 3.21 *For $n \geq 2f + 1$, the algorithm above solves the consensus problem for f omission failures within time $4(f+1)d + Cd$.*

Proof: For any given execution, let h be the first quiet phase. By Lemma 3.10, each nonfaulty process decides in some phase $r \leq h$, by time t_h. If $h = 0$ then by Lemma 3.10, each nonfaulty process decides in phase 0 in its first step and the running time is 0. If $h = 1$ then by Lemma 3.10, each nonfaulty process decides in phase 1 or 0, and by time t_1; by Lemma 3.18 the running time is $t_1 - t_0 \leq Cd + d$.

If $h > 1$, then we can bound the time for phases $1, \ldots, h-1$ by Lemma 3.20, and the time for phase h by Lemma 3.18. Thus we have

$$
\begin{aligned}
t_h - t_0 &= \sum_{r=1}^{h-1} (t_r - t_{r-1}) + (t_h - t_{h-1}) \\
&\leq \sum_{r=1}^{h-1} (3 + b_r)d + (Cd + d) && \text{(by Lemmas 3.20 and 3.18)} \\
&\leq (f+1)3d + f \cdot d + (Cd + d) && \text{(by Lemma 3.13 and Cor. 3.15)} \\
&= 4(f+1)d + Cd.
\end{aligned}
$$

∎

For $C \geq 4$, it is possible to construct an execution that takes exactly time $3d + 4(f-3)d + 3d + Cd + d$. In this execution, the first phase takes time $3d$, the following $f - 3$ phases take time $4d$, the penultimate phase takes $3d$ and and the last phase takes time $Cd + d$. Each of the phases taking $4d$ develops when all processes receive an $r - 1$ message at time t_{r-1} and all but one, p_{r-1}, advances to phase r. Process p_{r-1} decides on $r - 1 \bmod 2$ at t_{r-1} (before it receives the $r - 1$ messsage) and sends an r message to exactly one other process, p_{r+1}, which receives its acknowledgments for its $r-1$ message at time $t_{r-1}+2d$ and sends an r message to exactly $n - f$ processes. By time $t_{r-1} + 4d$, all processes receive an r message and, except for one process, p_r, advance to phase $r + 1$. In the following phase, at time $t_r + 4d$, process p_{r+1} decides (the processes to which p_{r+1} omitted an r message run slowly and do not detect its failure until $t_{r-1} + 2d + (Cd + d) = t_{r-1} + 7d = t_r + 3d$, so it is not shut down before then). Remaining details are left to the reader.

3.5.2 Bounds for $n \leq 2f$

When $n \leq 2f$, we are able to bound the running time of the algorithm in two ways, yielding one expression that depends on the ratio $\frac{f}{n-f}$ and another expression that depends on the square root of C. We will use Lemmas 3.14 ($B_r \cap B_{r'} = \emptyset$), 3.16 (the timeliness of acknowledgments), 3.18 (the time for any phase), and 3.19 (sequences of causal r messages), and Corollaries 3.15 (the sum of the b_r), and 3.17 (also regarding acknowledgments) the proofs of which did not rely on the relative values of n and f.

Bound dependent on $\frac{f}{n-f}$

This bound requires a lemma about the length of causal sequences of r messages more complicated than Lemma 3.19. Processes not in B_r must send an r message to $n - f$ processes but not necessarily to a nonfaulty process. We therefore are not able argue as for $n \geq 2f + 1$ that phase r ends very soon after a process not in B_r sends an r message. Nevertheless, disregarding processes in B_r for the moment, if it were true that a process could not get an acknowledgment from another process that already sent an r message, then it would take at most time $(\frac{f}{n-f})d$ before a nonfaulty process received an r message. Our algorithm does not exactly enforce this restriction on acknowledgments, but it does prevent a process from using acknowledgments received from a process that *previously omitted* an r message to it. We are thus able to derive a bound of $(3\frac{f}{n-f} + b_r + 4)d$ below in Lemma 3.23. This argument is most easily made by considering a directed graph on the faulty processors. Accordingly, for a given execution, define

- directed graph $G_r^f = (V_r^f, E_r^f)$ where
  $V_r^f = \{$all processes that fail during the given execution$\}$.
  $E_r^f = \{(i, j) : i$ sends an r message to j; $i, j \in V_r^f$; $i \neq j\}$.

- $\delta_r^f(i,j) =$ length of the shortest path in G_r^f from i to j, where $i,j \in V_r^f$.
- $S_r^{t_{r-1}} = \{i : i$ sends an r message by time $t_{r-1}\}$.
- $S_r^{nf} = \{i : i$ sends an r message to a nonfaulty process$\}$.

Claim 3.22 *If phase r is not quiet and no nonfaulty process decides in phase $r-1$, then there exist faulty processes $\alpha \in S_r^{t_{r-1}}$ and $\gamma \in S_r^{nf}$ such that there is a path in G_r^f from α to γ.*

Proof: Let γ be the first process to send an r message to a nonfaulty process. Process γ must be faulty: by the choice of γ, no process sends an r message to a nonfaulty process earlier than γ sends an r message and therefore if γ is nonfaulty then no process sends an r message to γ before it sends its r message. Therefore γ must decide in phase $r-1$, contradicting our assumption that no nonfaulty process decides in phase $r-1$. We conclude that $\gamma \in S_r^{nf}$. Note that γ sends an r message before any nonfaulty process does.

Let C_f be the nodes in G_r^f from which node γ is reachable (including γ) and let α be the process such that no process in C_f sends an r message before α does. It follows that α sends an r message no later than γ does. Because no nonfaulty process sends an r message before γ does, α does not receive an r message from a nonfaulty process before sending its r message. By choice, α does not receive an r message from any faulty process before it sends its r message. We therefore conclude that α receives no r messages before sending its own, and therefore must decide in phase $r-1$, sending an r message by time t_{r-1} (by its definition). ∎

Lemma 3.23 *For $r \geq 1$, if for all $r' \leq r$ phase r' is not quiet, then*
$$t_r - t_{r-1} \leq (3\tfrac{f}{n-f} + b_r + 4)d.$$

Proof: We show that by time $t_{r-1} + (3\frac{f}{n-f} + b_r + 4)d$, every process receives an r message. Thus, by that time, every process that is never blocked in any phase $r' \leq r$ and is neither decided nor shut down at that time, has $|A^{r-1}| \geq n-f$ by Lemma 3.16 (because it is not blocked in phase r) and therefore may advance to phase $r+1$.

First note that if a nonfaulty process decides in phase $r-1$, it does so by time t_{r-1}, broadcasting an r message that is subsequently received by all processes by time $t_{r-1}+d$, and the lemma is proved. So we consider the case that no nonfaulty process decides in phase $r-1$.

Lemma 3.22 applies for this case: it says that there exist processes $i \in S_r^{t_{r-1}} \cap V_r^f$ and $j \in S_r^{nf} \cap V_r^f$ such that j is reachable from i in G_r^f. Let $\alpha \in S_r^{t_{r-1}}$ and $\gamma \in S_r^{nf}$ be a closest pair of nodes in G_r^f:
$$\delta_r^f(\alpha,\gamma) = \min_{\substack{i \in S_r^{t_{r-1}} \cap V_r^f \\ j \in S_r^{nf} \cap V_r^f}} \delta_r^f(i,j).$$

We first bound the time by which γ broadcasts its r message. Fix a minimal length path from α to γ and let β be the last process on that path that sends an r message by time $t_{r-1} + 2d$. We claim that γ broadcasts its r message by time $t_{r-1} + (\delta_r^f(\beta, \gamma) + 2)d$. By the choice of β, each process j on the path from β to γ sends an r message later than time $t_{r-1} + 2d$ and therefore by Corollary 3.17 has $|A_j^{r-1}| \geq n - f$ by time $t_{r-1} + 2d$. By time $t_{r-1} + 3d$, the process on this path after β receives an r message (from β) and thus broadcasts an r message. Similarly, for $j \geq 1$, the j^{th} process on the path after β sends an r message by time $t_{r-1} + 2d + jd$ and process γ sends an r message by time $t_{r-1} + (2 + \delta_r^f(\beta, \gamma))d$.

We now show that by time $t_{r-1} + (\delta_r^f(\beta, \gamma) + 4)d$, all processes receive an r message. A nonfaulty process η receives an r message from γ by $t_{r-1} + (3 + \delta_r^f(\beta, \gamma))d$. Because no phase $r' \leq r$ is quiet, it follows from Lemma 3.4 that η does not decide in any phase $r' \leq r - 2$. If η decides in phase $r - 1$, then it does so by time t_{r-1}, sending an r message as it does; all processes receive it by time $t_{r-1} + d$. If η advances to phase r without deciding, then it does so by time t_{r-1}. By Corollaries 3.6 and 3.17, η has $|A_\eta^{r-1}| \geq n - f$ by time $t_{r-1} + (3 + \delta_r^f(\beta, \gamma))d$. By this time, η has received an r message from γ and therefore if η has not yet sent an r message—if η has not yet advanced from phase r or has decided in phase r and advanced to phase $r + 2$ but not yet sent an r message—it may then send an r message. An r message is then received by all processes by time $t_{r-1} + (\delta_r^f(\beta, \gamma) + 4)d$.

To complete the proof, we now show $\delta_r^f(\beta, \gamma) \leq \frac{3f}{n-f} + b_r$. Let $k = \delta_r^f(\beta, \gamma)$ and let $L_i = \{p : \delta_r^f(\beta, p) = i\}$ for $1 \leq i \leq k$. Define $L_0 = \{\beta\}$ and $L_{-1} = \emptyset$. Consider the sum

$$\sigma = \sum_{i=0}^{k-1} |L_{i-1} \cup L_i \cup L_{i+1}|.$$

Since the sets L_i are disjoint, each node in G_r^f is counted at most 3 times, so$\sigma \leq 3|G_r^f| \leq 3f$.

Claim 3.24 *For $0 \leq i \leq k - 1$, at least $k - b_r$ of the sets $L_{i-1} \cup L_i \cup L_{i+1}$ has cardinality at least $n - f$.*

Proof: Clearly, for $i \leq k - 1$, no set L_i is empty, since γ, at distance k from β, receives an r message from a faulty process. At least $k - b_r$ sets L_i contain a process $j \notin B_r$ such that j is on the path from β to γ. For each j, and each process ℓ in A_j^r, clearly, j sends ℓ an r message; we will show that if j is on the chosen path from β to γ, then process ℓ sends j an r message also. We will also show that if $j \in L_i$ where $i \leq k - 1$, then ℓ is faulty and therefore in G_r^f. Thus, for all ℓ in A_j^r such that $j \in L_i$, there are edges of G_r^f in both directions between j and ℓ and if $j \notin B_r$, then $|L_{i-1} \cup L_i \cup L_{i+1}| \geq n - f$, completing the proof.

We first show that if $j \in L_i$ where $i \leq k - 1$, then $\ell \in A_j^r$ is faulty. If ℓ were nonfaulty, then j would be in S_r^{nf}. But j cannot be in S_r^{nf}, since γ, at distance k, was defined to be the closest node to α in $S_r^{nf} \cap V_r^f$ but $j \in L_i$ is at distance $i < k$.

We next show that for each j on the chosen path from β to γ, if $\ell \in A_j^r$, then ℓ sends j an r message. Consider first the case that ℓ broadcasts an r message before sending "ack(j,r)" to j. Since $\ell \in A_j^r$, ℓ does not omit a message to j before sending "ack(j,r)"; in particular, it does not omit the r message. Consider then the case that ℓ does not send an r message before sending "ack(j,r)" to j. By the choice of β, j sends its r message (and ℓ receives it) later than time $t_{r-1}+2d$. Because ℓ sends an "ack(j,r)" message, ℓ either decides in phase $r-1$ or advances to phase r without deciding. However, ℓ cannot decide in phase $r-1$: processes that decide in phase $r-1$ do so, sending an r message by time t_{r-1}, but we are assuming ℓ does not send an r message before sending "ack(j,r)", which is later than time $t_{r-1}+2d$. Thus, at the time that ℓ receives the r message from j, ℓ is either in phase r, not yet having sent an r message, or decided and in phase $r+2$, not yet having sent an r message. In its first step after receiving the r message from j, process ℓ queues both "r" and "ack(j,r)". Because $\ell \in A_j^r$, this message is not omitted, so ℓ sends an r message to j. ■

Thus we have $(k-b_r)(n-f) \le \sigma$ and $\sigma \le 3f$, or $\delta_r^f(\beta,\gamma) = k \le \frac{3f}{n-f} + b_r$, which completes the proof: all processes receive an r message by time $t_{r-1} + (\frac{3f}{n-f} + b_r + 4)d$. ■

Theorem 3.25 *For $n \le 2f$, the algorithm above solves the consensus problem for f omission failures within time $(3\frac{f}{n-f}+5)(f+1)d + Cd$.*

Proof: For any given execution of the algorithm in which h is the first quiet phase, by Lemma 3.10, each nonfaulty process decides in some phase $r \le h$, by time t_h. Again, if $h=0$ then by Lemma 3.10, each nonfaulty process decides in phase 0 in its first step and the running time is 0. If $h=1$ then by Lemma 3.10, each nonfaulty process decides in phase 1 or 0, and by time t_1; by Lemma 3.18 the running time is $t_1 - t_0 \le Cd + d$.

If $h > 1$, we can bound the time for phases $1,\ldots,h-1$ by Lemma 3.23, and the time for phase h by Lemma 3.18. Thus we have

$$\begin{aligned}
t_h - t_0 &= \sum_{r=1}^{h-1}(t_r - t_{r-1}) + (t_h - t_{h-1}) \\
&\le \sum_{r=1}^{h-1}(4 + 3\tfrac{f}{n-f} + b_r)d + (Cd+d) && \text{(by Lemmas 3.23 and 3.18)} \\
&\le (f+1)(4+3\tfrac{f}{n-f})d + f\cdot d + (Cd+d) && \text{(by Lemma 3.13 and Cor. 3.15)} \\
&= (f+1)(5+3\tfrac{f}{n-f})d + Cd.
\end{aligned}$$

■

Bound dependent on $\sqrt{C}$

The analysis of the previous section shows that the running time of our algorithm is

$$(5 + 3\tfrac{f}{n-f})(f+1)d + Cd.$$

Note that if f is close to n, say $n = f+1$, then the bound is roughly proportional to $f^2 d$, which is no improvement on the algorithm of [ADLS90]. However, for these proportional values of n and f, we are better able to bound the running time. The analysis of this section shows that the running time of our algorithm is also bounded by

$$(2\sqrt{C} + 6)(f+1)d + Cd.$$

So when $2\sqrt{C} + 6 < 5 + 3\frac{f}{n-f}$, or roughly $n < f + f/\sqrt{C}$, we have a better bound on the running time.

Define a partition the first r phases of the given execution into two classes according to their length:

$$\begin{aligned} X_r &= \{\rho : t_\rho - t_{\rho-1} \leq \sqrt{C} \cdot d \text{ and } \rho \leq r\} = \{\text{short phases}\} \\ Y_r &= \{\rho : \rho \notin X_r \text{ and } \rho \leq r\} = \{\text{long phases}\}. \end{aligned}$$

and define

$$S_r^f = \{i : \; i \text{ omits an } r \text{ message to a nonfaulty process after time } t_{r-1}\}.$$

We can bound the short phases by their defined bound, but bound the long phases by chains of r messages, via the following two lemmas.

Lemma 3.26 *If phase $r \geq 1$ is not quiet then either $t_r \leq t_{r-1} + (|S_r^f| + 3)d$ or all nonfaulty processes decide by this time.*

Proof: We once again show that by time $t_{r-1} + (|S_r^f| + 3)d$, all processes receive an r message and thus by this time are either decided, shut down, blocked in some phase $r' \leq r$, or may advance to phase $r+1$.

By Lemma 3.19, we know that for any process i_0, there is a sequence of distinct processes $i_0, i_1, \ldots, i_k$ such that k is the only process in the sequence to broadcast an r message before time t_{r-1}, and for $0 < j \leq k$, by time $t_{r-1} + (1+k-j)d$, i_j sends the first r message, m_{j-1}, received by i_{j-1}.

Now, if $k \leq |S_r^f|$ then i_0 receives m_0 by time $t_{r-1} + (k-1+1)d + d \leq t_{r-1} + (|S_r^f|+1)d$. If $k > |S_r^f|$, then there is a j such that $0 < k - |S_r^f| \leq j \leq k$ and $i_j \notin S_r^f$. Process i_j therefore sends an r message to all nonfaulty processes by time $t_{r-1} + (1+k-j)d \leq t_{r-1} + (1+|S_r^f|)d$.

If some nonfaulty process has not yet decided, then it sends an r message to all other processes by time $t_{r-1} + (2 + |S^f_r|)d$ and process i_0 receives an r message by time $t_{r-1} + (3 + |S^f_r|)d$. ■

The key observation is that a process cannot fail to a nonfaulty process in many long phases:

Lemma 3.27 *For any execution of the protocol taking at least ϕ phases and for any process j, there are at most $\sqrt{C} + 3$ phases $\rho \in Y_\phi$ such that $j \in S^f_\rho$.*

Proof: If j omits an k message to a nonfaulty process at time t then by Lemma 3.1, that nonfaulty process detects j's failure by time $t + Cd + d$, broadcasting "shutdown j" at that time. We have $t \le t_k$, and so j is shut down by time $t + Cd + 2d \le t_k + (\sqrt{C} + 2)\sqrt{C}d$. It follows that there are at most $\sqrt{C} + 2$ long phases ℓ such that $t_k < t_\ell \le Cd + 2d$. Thus j cannot attempt to send (and cannot omit) an $\ell + 1$ message after time t_ℓ and is therefore not in S^f_ρ for any $\rho > \ell$. ■

Theorem 3.28 *For $n \le 2f$, the algorithm above solves the consensus problem for f omission failures within time $(2\sqrt{C} + 6)(f + 1)d + Cd$.*

Proof: Let phase h be the first quiet phase. Again, if $h = 0$ then by Lemma 3.10, each nonfaulty process decides in phase 0 in its first step and the running time is 0. If $h = 1$ then by Lemma 3.10, each nonfaulty process decides in phase 1 or 0, by time t_1; by Lemma 3.18 the running time is $t_1 - t_0 \le Cd + d$.

If $h > 1$, we consider two cases. Consider first the case that not all nonfaulty processes decide in phase $h - 2$. We bound the length of the short phases by their defined length. We bound the length of the long phases by Lemma 3.26 and then sum the sizes of S^f_ρ using Lemma 3.27. The length of phase h is bounded by Lemma 3.18. Thus we have

$$
\begin{aligned}
t_h - t_0 &= \sum_{\rho \in X_{h-1}} (t_\rho - t_{\rho-1}) + \sum_{\rho \in Y_{h-1}} (t_\rho - t_{\rho-1}) + (t_h - t_{h-1}) \\
&\le |X_{h-1}| \cdot \sqrt{C}d + d \sum_{\rho \in Y_{h-1}} (3 + |S^f_\rho|) + (Cd + d) && \text{(by Lemmas 3.26 and 3.18)} \\
&\le |X_{h-1}| \cdot \sqrt{C}d + 3|Y_{h-1}|d + f(\sqrt{C} + 3)d + (Cd + d) && \text{(by Lemma 3.27)} \\
&\le (f+1)\sqrt{C}d + 3(f+1)d + f(\sqrt{C} + 3)d + Cd + d && \text{(by Lemma 3.13)} \\
&< (2\sqrt{C} + 6)(f+1)d + Cd.
\end{aligned}
$$

Now consider that case that all nonfaulty processes decide in phase $h-2$. The running is then bounded by $t_{h-2} - t_0$:

$$\begin{aligned}
t_{h-2} - t_0 &= \sum_{\rho \in X_{h-2}} (t_\rho - t_{\rho-1}) + \sum_{\rho \in Y_{h-2}} (t_\rho - t_{\rho-1}) \\
&\le |X_{h-2}| \cdot \sqrt{C} d + \sum_{\rho \in Y_{h-2}} (3 + |S_\rho^f|) d && \text{(by Lemma 3.26)} \\
&\le |X_{h-2}| \cdot \sqrt{C} d + 3|Y_{h-2}| d + f(\sqrt{C} + 3) d && \text{(by Lemma 3.27)} \\
&\le f\sqrt{C} d + 3fd + f\sqrt{C} d + 3fd && \text{(by Lemma 3.13)} \\
&= (2\sqrt{C} + 6) fd
\end{aligned}$$

■

Chapter 4

Consensus in the Presence of Byzantine Failures

In this chapter we present a simulation algorithm using $3f + 1$ processes and tolerating f Byzantine failures. The algorithm simulates any synchronous round-based algorithm tolerant of f Byzantine failures and uses time $r(d + 2Cd) + Cd$, where r is the number of rounds required by the synchronous algorithm.

The simulation works by keeping processes loosely synchronized to ensure that a nonfaulty process does not advance to round r until it has received a round $r - 1$ message from every nonfaulty process. The partial synchronization works by using a combination of two criteria for advancing to further phases, one based on elapsed local time and the other based on messages received. A similar technique is used in [WL88] to initiate new rounds of clock resynchronization. In particular, our criteria for ending round 1 is essentially the same as the criteria used in [WL88] for ending every round; our criteria for subsequent rounds is different.

4.1 The simulation algorithm

The algorithm simulates a synchronous algorithm by ensuring that each nonfaulty process receives all round r messages of the synchronous algorithm from all other nonfaulty processes before advancing to round $r + 1$. We do not explore here the formal semantics of "a correct simulation"; rather we regard as sufficient the following correspondence ensured by the above property: For every execution of the simulation, there is an execution of the round-based synchronous algorithm in which the nonfaulty processes receive the same vector of messages from each other at each round. Since the behavior of faulty processes is not restricted, clearly the corresponding synchronous execution is legal.

Therefore, for the purposes of simulation, we define a synchronous algorithm by its message function only, suppressing information about the state of the synchronous algorithm. Let $M_i(r, V^{r-1})$ denote the vector of messages to be sent in the synchronous algorithm by process i in round r when the ordered set of messages V^{r-1} is received in round $r-1$ (of course, this message function may also depend on the state of the process; we leave this implicit). Without loss of generality, assume each process sends a message to all processes at every round of the synchronous algorithm.

Recall that we assume all processes begin executing the algorithm at the same time. At each step, a process increments a counter s (initially 0) and executes the code in Figure 4.1, explained below. A local variable, initially 1, keeps track of the ROUND number. Ordered set V^r contains the r^{th} message received from each process. We refer to the r^{th} message sent by a process as a "round r" message. (Recall that we assume each process sends a message to all processes in every round of the synchronous algorithm.)

Each process first sends its round 1 message and then waits for at least time d to ensure that it receives a round 1 message from every other nonfaulty process. When it can be sure that time d has elapsed, it advances to round 2 and broadcasts its round 2 message based on the round 1 messages it has received so far. It ensures that time d has passed by either waiting for d/c_1 of its own steps or by receiving $f+1$ round 2 messages—this ensures that some nonfaulty process has waited at least time d.

In subsequent each round r, a process waits for at least time $2d$ (actually $2d+3c_2$) after at least $f+1$ *nonfaulty* processes have sent a round r message. By this time, all nonfaulty processes must have received at least $f+1$ round r messages and therefore advanced to round r and sent a round r message. At this time, a process advances to round $r+1$ and broadcasts its round $r+1$ message. Again, there are two ways for a process to deduce that sufficient time has passed: if it takes $(2d+3c_2)/c_1$ steps after receiving at least $2f+1$ round r messages or if it receives at least $f+1$ round $r+1$ messages. The latter ensures that some nonfaulty process has advanced to round $r+1$ and therefore has already waited a sufficient amount of time (at least time $2d$ after at least $f+1$ nonfaulty processes sent a round r message).

4.2 Correctness

Let t_r be the latest time that any nonfaulty process sends a round r message. Again, we assume that all processes begin at the same time (here, t_1). We say a process "advances

ROUND 1	Send $M(1,\cdot)$; goto ROUND $1'$.
ROUND $1'$	**If** $s > d/c_1$ or $\|V^2\| \geq f+1$,
	then goto ROUND 2
ROUND r	Send $M(r, V^{r-1})$; goto ROUND r'.
ROUND r'	**If** $\|V^r\| \geq 2f+1$,
	then $s \leftarrow 0$; goto ROUND r''.
ROUND r''	**If** $s > (2d+3c_2)/c_1$ or $\|V^{r+1}\| \geq f+1$,
	then goto ROUND $r+1$.

Figure 4.1: The simulation of a synchronous algorithm. At each step, a process increments the counter s and executes the code according to its present round number. V^r is the ordered set consisting of the r^{th} message received from each process. $M(r, V^{r-1})$ denotes the message function of the synchronous algorithm for round r.

to round r" when it executes the "goto ROUND r" statement in the code. In order to prove correctness, we must show that a nonfaulty process eventually advances to all rounds required by the synchronous algorithm and always receives a round r message from all nonfaulty processes before advancing to round $r+1$.

Lemma 4.1 *Each nonfaulty process advances to all rounds required by the synchronous algorithm.*

Proof: By induction on the round number. Clearly each nonfaulty advances to round 2—it advances to round $1'$ after its first step and advances to round 2 after at most $1 + d/c_1$ more steps.

For $r > 2$, assume all nonfaulty processes have advanced to round r. Then all nonfaulty processes have sent a round r message and advanced to round r'. Since $n - f \geq 2f+1$, there are at least $2f+1$ nonfaulty processes, so each nonfaulty process eventually receives at least $2f+1$ round r messages and advances to round r''. After at most $(2d+3c_2)/c_1$ steps in round r'', each nonfaulty process advances to round $r+1$. ∎

Lemma 4.2 *No nonfaulty process advances to round $r+1$ before receiving a round r message from each nonfaulty process.*

Proof: By induction on the round number r.

$r = 1$: Each nonfaulty process takes more than d/c_1 steps before advancing to round 2. Thus each advances later than time $t_1 + d$, which is after the round 1 message of each nonfaulty process, sent at time t_1, is delivered.

$r > 1$: Assume the lemma is true for $r - 1$ (i.e., no nonfaulty process advances to round r before receiving a round $r-1$ message from each nonfaulty process). We show the lemma is true for r. Let i be the first correct process to advance to round $r + 1$ and let τ_i be the time at which i advances to round r'' (by Lemma 4.1, this time is well-defined). We make the following series of deductions about the events that occur at or before the listed times:

τ_i : Because i is in round r'', by the induction hypothesis, i has received a round $r - 1$ message from all nonfaulty processes. Because i has advanced to r'', it has received at least $2f + 1$ round r messages.

$\tau_i + d$: All nonfaulty processes are in round $(r - 1)'$ or greater (because they have each sent an $r - 1$ message to i) and have received at least $2f + 1$ round $r - 1$ messages (from each other).

$\tau_i + d + c_2$: All nonfaulty processes therefore advance to round $(r - 1)''$.

$\tau_i + d + 2c_2$: All nonfaulty processes have received at least $f + 1$ round r messages (from the nonfaulty subset of processes that sent round r messages to i) and therefore advance to round r.

$\tau_i + d + 3c_2$: All nonfaulty processes send a round r message and advance to round r'.

$\tau_i + 2d + 3c_2$: All processes receive a round r message from each nonfaulty process.

Because by choice i is the first nonfaulty process to advance to round $r + 1$, it follows that i advances to round $r+1$ only after $(2d + 3c_2)/c_1$ steps in round r'', which occurs later than time $\tau_i + 2d + 3c_2$. We conclude that i receives a round r message from each nonfaulty process before advancing to round $r+1$. Since all nonfaulty processes advance to round $r+1$ after i, they also receive a round r message from all nonfaulty processes before advancing. ■

4.3 Analysis of time bounds

Again, we assume $d \gg c_2$ and therefore approximate $d + c_2$ by d in the timing analysis.

Lemma 4.3 $t_2 - t_1 \leq Cd$ *and, for* $r \geq 2$, $t_{r+1} - t_r \leq d + 2Cd$.

Proof: Clearly, $t_2 \leq t_1 + Cd$. By time t_r all nonfaulty processes send a round r message and advance to round r'. Therefore by time $t_r + d$, all nonfaulty processes receive at least $2f + 1$ round r messages and advance to round r''. Within another time $2Cd$, all nonfaulty processes have taken $(2d + 3c_2)/c_1$ steps and advanced to round $r + 1$, sending an $r + 1$ message. ■

Theorem 4.4 *There is an algorithm using* $3f + 1$ *processes which solves the consensus problem for* f *Byzantine failures within time* $Cd + f(d + 2Cd) = fd + (2f + 1)Cd$.

Proof: Any $(f+1)$-round synchronous algorithm can be simulated. Agreement and validity follow from correct simulation. Termination follows from Lemma 4.1. The time bound follows from Lemma 4.3. ■

Chapter 5

Bounded-capacity Message Links and Failure Detection

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. In this chapter, we consider how quickly such failures can be detected in our semi-synchronous model.

If it is assumed that all messages sent are delivered within time d of when they are sent, then the following simple protocol minimizes the time between any failure and its detection. (This is the strategy employed in the algorithm of [ADLS90] and our algorithm of Chapter 3.) Each processor broadcasts a message at every step that it takes. If no message is received from another processor for more than $(d + c_2)/c_1$ local steps, that processor is declared faulty. Because local steps are separated by at least time c_1, at least time $d + c_2$ passes before this many steps are taken. Because local steps are separated by at most time c_2, the time between the delivery of any two consecutive messages sent by a processor is at most $d + c_2$. It follows that only failed processors are declared faulty. The maximum time between any failure and its detection is approximately $Cd + d$, occurring in the following scenario: processor p broadcasts a message at time t and then fails; these messages are delivered at time $t + d$; every other processor runs slowly (its steps separated by c_2) after $t + d$, and thus p's failure is not detected until time $t + d + c_2(d + c_2)/c_1 \approx t + d + Cd$.

Although the above protocol guarantees minimal delay between any failure and its detection, it is clearly inefficient in its use of messages. It takes advantage of the strong assumption that all messages are delivered within time d, regardless of the rate at which they are sent. In reality, the performance of a message link may suffer if messages are sent too frequently. In this chapter, we propose a model of message links with bounded capacity and analyze the effect of the capacity bound on the efficiency of detecting stopping failures.

5.1 Modeling bounded-capacity links

Define a *message link of unit capacity and delay d* as a communication channel that queues incoming messages in FIFO order and delivers the first message in the queue within time d of the later of when the message is sent and when the previous message is delivered. (For simplicity, we will assume that message links deliver messages in the order sent. Our algorithms do not make use this assumption and our lower bounds hold in spite of it.) For positive integer μ, define a *message link of capacity μ and delay d* as the composition of μ message links each of unit capacity and delay d/μ, connected serially so that messages are delivered from link i to link $i+1$ for $1 \leq i < \mu$ and link μ delivers messages to the recipient process. Messages are neither lost by a link nor delivered out of order, and once a processor has sent a message, it cannot cancel that message.

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 the 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.

On the other hand, 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 an induction on the number of messages sent. Assume the previous message is delivered by the i^{th} sublink within time $i \cdot d/\mu$ of when it is sent (clearly this is true for the "first" message ever sent). If message m is sent at time t, then for $0 \leq i < \mu$, by time $t + i \cdot d/\mu$ the previous message has been delivered by link $i+1$ and m is delivered by link i (by induction on i). Thus m is delivered to the recipient process within time $\mu \cdot d/\mu = d$ of when it is sent.

For the lower bound, we assume only that in the worst case, a link delivers every pair of messages at least time d/μ apart.

5.2 Timing out failed processors

We will consider a system of processors fully connected by message links of capacity μ and delay d. These processors may fail by stopping. A process 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.

Because in our model each pair of processors is connected by a private bidirectional message link, we will assume that the timeout protocol executes independently for each pair of processors. We will therefore prove bounds on detection time for a system of two processors, p and q.

5.2.1 Upper bounds on detection time

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 a token message back to q, and q does likewise. If a processor takes more than $(2d+c_2)/c_1$ 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, $d + C(2d + c_2) + c_2$; 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 are delivered to q and q has sent its last message to p; within another time $c_2(1 + (2d + c_2)/c_1) \approx 2Cd$, q has taken enough steps to conclude that p has failed.

An upper bound of $C^2d/\mu + Cd + d$ is achieved by a protocol in which each processor sends a message every $(d/\mu)/c_1$ steps. A process concludes that the other has failed if it has taken 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 5.1, 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 then $c_2(d/\mu)/c_1 + d = Cd/\mu + d$ (if the first message is delivered immediately 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 $\min(2Cd + d,\ C^2d/\mu + Cd + d)$. Note that $2Cd + d < C^2d/\mu + Cd + d$ if and only if $\mu < C$.

5.2.2 Lower bounds on detection time

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 $\mu < C+1$: when $C \leq \mu < C+1$, $C^2d/\mu + Cd + d$ is the best upper bound and $2Cd + d/\mu$ is the best lower bound; 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 some pair of consecutive messages from p to q are separated by at least time d/μ. We prove that such an execution exists for any protocol 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 (put onto the link) faster than they were delivered (removed from the link). Thus the number of messages sent but undelivered and, consequently, the total delay of a message, would grow in time without bound. The time between when p fails and when q receives no further messages is therefore increased without bound.

Lemma 5.1 *For all B and for any correct timeout protocol that guarantees a detection time of B, there exists an execution in which*

1. *All consecutive steps of p are separated by c_1;*
2. *All consecutive steps of q are separated by c_2;*
3. *All messages from q to p are delayed by time 0;*
4. *All messages from p to q are delayed by at least time d;*
5. *For all t_0, there exists a pair of messages m_1 and m_2 sent by p at times t_1 and t_2 respectively, such that $t_1 \geq t_0$, $t_2 - t_1 \geq d/\mu$, and no message is sent by p in the interval (t_1, t_2).*

Proof: For contradiction, suppose not. 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 the fourth timing constraint is satisfied—all messages from p to q are delayed at least time d. We prove that the fifth condition is also satisfied in β. To do so, assume for contradiction that it is not.

First note that because β is infinite, p must send an infinite 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 during 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 infinite number of messages and, by assumption, every two consecutive messages are separated at most time $k \cdot c_1$, process p sends at least $\lfloor x/(k \cdot c_1) \rfloor$ 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 $(\frac{x}{k \cdot c_1} - 1) - (\frac{x}{d/\mu} + 1)$. 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 $t_0 + x + \frac{d}{\mu}(\frac{x}{k \cdot c_1} - \frac{x}{d/\mu} - 2)$. Let x be any number large enough so that $\frac{d}{\mu} \cdot (\frac{x}{k \cdot c_1} - \frac{x}{d/\mu} - 2) > B$ (recall that $k \cdot c_1 < d/\mu$).

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. This is a contradiction on the assumed protocol. ∎

Our lower bound proof uses the retiming techniques of "shifting" events in time and "shrinking" portions of executions that were developed in [AL89] and [LL84].

Theorem 5.2 *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* $\min(2Cd + d/\mu,\ C^2 d/\mu + Cd + d)$.

Proof: Let $T = \min(2Cd + d/\mu,\ C^2 d/\mu + Cd + d)$. 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 5.1 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

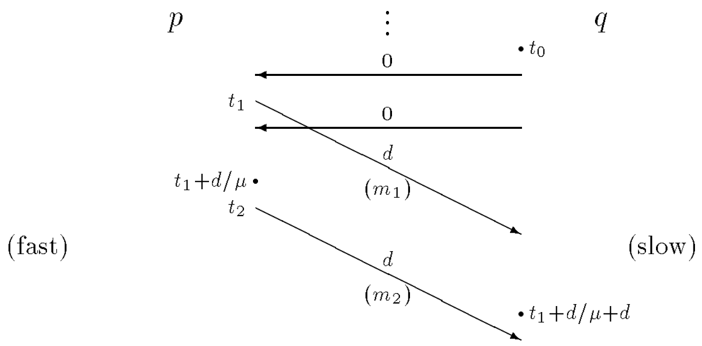

Figure 5.1: Execution β, the existence of which is proved by Lemma 5.1, takes the above form except that messages from p to q may be delayed more than d and messages may be sent by q at arbitrary times. The events of p (q) are on the left (right), with time represented by the vertical dimension. An arrow represents a message labelled with its delay, with its tail at the time of the send event and its tip at the time of the receive event.

at times t_1 and t_2 respectively. Figure 5.1 depicts an example of an execution satisfying Lemma 5.1; for presentation, messages from p to q are shown taking exactly time d and messages from q to p are shown 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 messages from p to q are delayed by at least time d in β and so q doesn't receive m_2 until $t_2 + d \geq t_1 + d/\mu + d$. 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 retiming the events of α and β, we construct executions α' and β', which are indistinguishable to both p and q from α and β respectively. By retiming the events of α', we construct α'', which is indistinguishable from α' to q. Finally, by retiming the events of β', we construct β'', which is indistinguishable from β' to p up to the time that it sends m_2 and indistinguishable from α'' to q up to the time that it times out p in α'' Thus, although p does not fail in β'', q times out p in β'', contradicting the correctness of the assumed protocol.

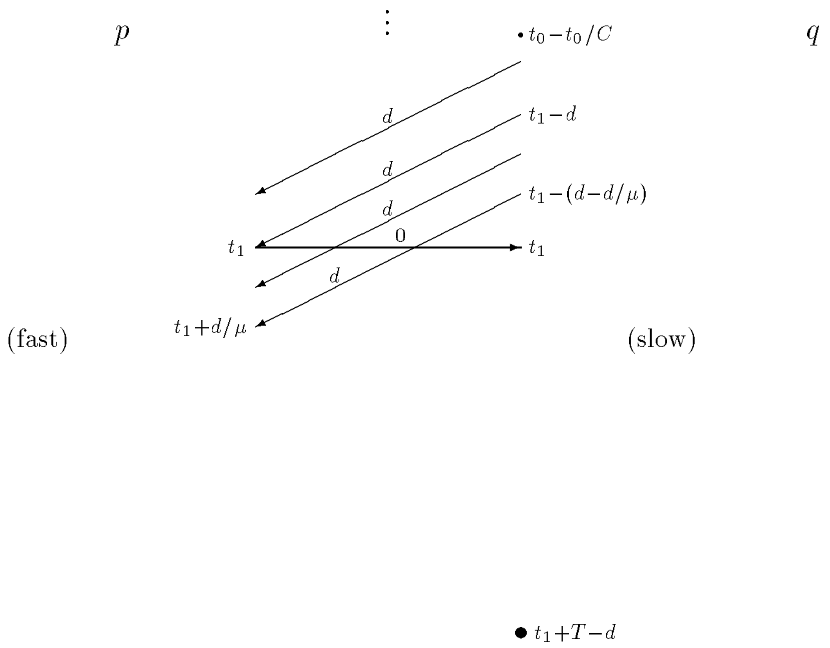

Figure 5.2: In the region of interest, execution α' is simply α with events of processor q occurring earlier in time by d. Because p fails at time t_1, q detects the failure of p by time $t_1 + T - d$, denoted by the circle.

Construction of executions 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). Strictly speaking, this may not be possible for all events at q because of initial conditions. However, it is sufficient to shift by d the events of q that occur after time t_1 in α and "shrink" some interval before t_1 in α (i.e., retiming the events of the inverval so that q runs fast in that interval of events in α'). In particular, we shrink the interval $[0, \frac{C}{C-1}d]$. Note that by our choice of $t_0 = \frac{C}{C-1}d$ in choosing β, the last event of this interval occurs before m_1 is sent at time t_1. Leaving all events at time 0 unchanged, steps of q in this interval, which take time c_2 in α, are retimed to take time c_1 in α'. Thus the interval is shrunk by a factor of C and the last event of the interval occurs earlier in α' by $\frac{C}{C-1}d - \frac{1}{C-1}d = d$. Figure 5.2 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 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' (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'.

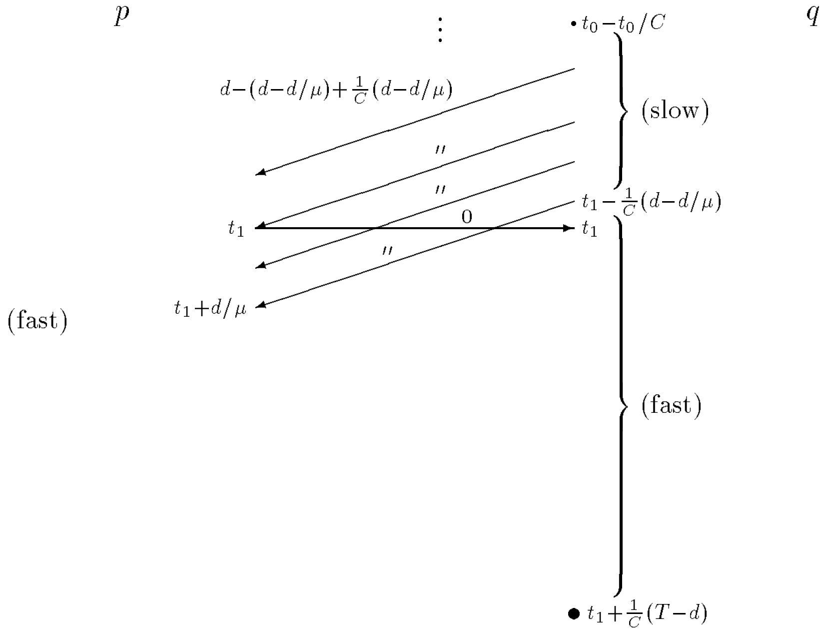

Figure 5.3: Execution α'' is constructed from α' by mapping the interval $[t_1 - (d-d/\mu),\ t_1+(T-d)]$ of α' to the interval $[t_1 - \frac{1}{C}(d-d/\mu),\ t_1 + \frac{1}{C}(T-d)]$ of α'' and appropriately shifting the rest of q's events.

Construction of execution α''

Recall that q runs slowly in α and most of α'—its steps are separated by c_2. We now construct α'' from α' by a 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; after time t_1, the events (message deliveries) at p 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''. 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 merely to preserve the step times of events on the borders of this interval. To be precise, α'' is defined at q for 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:

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

This execution is illustrated in Figure 5.3.

> 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'.

By construction, this retiming operation does not cause violations of the bounds on process 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 $t'' \le t_1$ and $t \ge \frac{C}{C-1}d$, we have $t - d \le t'' < t$. By the retiming mapping above, as a function of t' we have

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

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

$$t - d \;\le\; t'' \;<\; t - d/\mu - \frac{1}{C}(d - d/\mu) \;<\; t. \tag{5.1}$$

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 $t'' \le t_1$ in α'' then by Equation 5.1, it is delivered earlier in α'' than in α by not more than d; it follows that the message 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 5.1 they are sent earlier (and delayed more) by not more than d. Any message sent by q after t_1 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''.

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 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 received at q at least time d later.

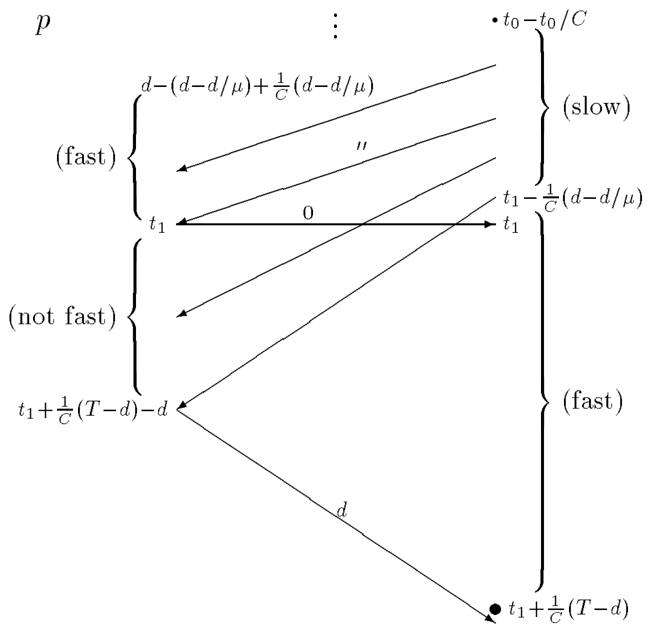

Figure 5.4: Execution β'' is essentially the same as execution α'', except that p does not fail; instead, it runs slowly after sending message m_1, and message m_2 is delayed by d. Because p sends no other messages before m_2, this execution appears the same as α'' to q until it receives m_2.

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 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 for $t'' \le t_1 + \frac{1}{C}(T-d) - d$ by retiming each each event that occurs at time t' in β' to occur at time t'' in β'', where t'' is defined as follows:

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

This execution is illustrated in Figure 5.4.

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 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.

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 $t_1 \le t_i \le t_j \le t_1 + d/\mu$, we have $t''_j - t''_i \le C(t_j - t_i)$:

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

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

We now verify that messages sent by p after t_1 are within the proper bounds. The first message sent by p after t_1 is m_2, which in β' (and β) is sent at $t_2 \ge t_1 + d/\mu$ and thus in β'' is sent at $t''_2 \ge t_1 + \frac{1}{C}(T-d) - d$. 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 $t_1 + \frac{1}{C}(T-d)$. (Note that q times out p by this time.) 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 $t' \le t_1 - d$ in β'.

A message sent by q at time $t_1 - d$ in β' is sent and delivered at time t_1 in β. Since the events at p are the same in β'' and β and α'', messages sent by q before $t_1 - d$ in β'' are delivered to p by t_1 in β'' and are therefore, by the analysis of α'', are delayed by correct amounts.

Case 2: q sends at time t' in β' where $t_1 - d \le t' \le t_1 - (d - d/\mu)$.

Such a message is delivered at time $t' + d$ in β' where $t_1 \le t' + d \le t_1 + d/\mu$. In β'' the sending event at q is mapped to time $t' + (d - d/\mu) - \frac{1}{C}(d - d/\mu)$, which is less than t_1. 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, this messages is delayed by at least 0. We show below that it is delayed by at most d:

$$\begin{aligned}
& t_1 + \frac{\frac{1}{C}(T-d) - d}{d/\mu}(t' + d - t_1) - \left[t' + (d - d/\mu) - \frac{1}{C}(d - d/\mu)\right] \\
= \;& t_1 + \left(\frac{\frac{1}{C}(T-d) - d}{d/\mu} - 1\right) t' + \left(\frac{\frac{1}{C}(T-d) - d}{d/\mu}\right)(d - t_1) - \left[(d - d/\mu) - \frac{1}{C}(d - d/\mu)\right] \\
\le \;& t_1 + \frac{\frac{1}{C}(T-d) - d}{d/\mu}(t_1 - (d - d/\mu) + d - t_1) - \left[t_1 - (d - d/\mu) + (d - d/\mu) - \frac{1}{C}(d - d/\mu)\right] \\
& \text{since } t' \le t_1 - (d - d/\mu) \text{ and } \left(\frac{\frac{1}{C}(T-d) - d}{d/\mu} - 1\right) \ge 0 \\
\le \;& t_1 + \frac{1}{C}(T-d) - d - \left[t_1 - \frac{1}{C}(d - d/\mu)\right] \\
\le \;& \frac{1}{C}(2Cd + d/\mu - d) - d + \frac{1}{C}(d - d/\mu) \qquad \text{since } T \le 2Cd + d/\mu \\
= \;& d.
\end{aligned}$$

Case 3: q sends at time $t' \ge t_1 - (d - d/\mu)$ 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

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

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. ∎

5.2.3 Bounds for two processors using a single message link

We remark that our techniques give 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 5.2.1 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 5.2 specializes to the unidirectional case to give a corresponding lower bound of $C^2d/\mu + Cd + d$. Theorem 5.2 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 5.1, 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 5.2 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 $C^2d/\mu + Cd + d$.

5.3 Consensus with bounded-capacity message links

We remark on how our upper bounds for consensus are affected by bounding the capacity of the message links used.

5.3.1 Byzantine failures

Because it is not message-intensive, our algorithm for Byzantine failures is not affected by the restriction of bounded-capacity message links. Recall that the algorithm for Byzantine failures does not include a fault-detection task and does not require a process to send a message at every step it takes. The correctness follows from the fact that at least time $2d$ passes between the time that the round $r-1$ messages of all nonfaulty processes are delivered and the time that any nonfaulty process advances to round $r+1$. The round r message of a nonfaulty process can incur more than delay d only if it is sent before the previous message is delivered. The previous message is its round $r-1$ message, so even if the round r message incurs added delay, it is still delivered by time $2d$ (actually, d) after the round $r-1$ messages of all nonfaulty processes are delivered, and all nonfaulty processes receive it before advancing to round $r+1$. Otherwise, if the round r message does not incur added delay due to the capacity of the message link, the proof of Lemma 4.2 holds as before. Because nonfaulty processes do not send any messages other than the messages of the synchronous algorithm, it is easy to see that the delay of messages does not affect the proof of Lemma 4.3, and the running time is not affected.

5.3.2 Omission failures

First note that if every pair of messages sent by a process are separated by at least time d/μ, then each message is delayed by at most time d and the omissions algorithm is not affected (the analysis of Chapter 3 holds). Because the fault-detection protocol requires a process to send a message at every step it takes, messages may be separated by as little as time c_1; therefore the omissions algorithm is not affected if the message links are of capacity $\mu \geq d/c_1$.

The first consequence of using links of capacity $\mu < d/c_1$ is the obvious effect on the fault-detection protocol. Instead of the bound $Cd + d$ guaranteed for the time until a failure is detected (shown in Lemma 3.1), a bound of only $\min(2Cd + d/\mu,\ C^2 d/\mu + Cd + d)$ can be guaranteed by the fault-detection protocol. Lemmas 3.7 and 3.18 then also involve the above expression instead of $Cd + d$.

But a more serious affect on the running time of the algorithm is the added delay between when a process "should" send a message (according to the main algorithm) and when it *may* send it. A crucial element of the algorithm is the piggybacking of messages of fault-detection task and messages of the main algorithm. A straightforward implementation would require that messages of the main algorithm can only be sent during steps in which a message of the fault-detection task is to be sent.

If the first timeout task, in which each pair of processes continually sends a "token" back and forth, were used for the fault-detection protocol, up to time $2d$ may elapse between when a process is required to send a message of the main algorithm and when it is able to piggyback that message onto a message of the fault-detection task. Thus each message may in effect be delayed by a total of $3d$, since the timeout task ensures that all messages are delivered within time d of when they are sent, despite the capacity of the message links. This gives bounds of

$$\begin{array}{rll} 4(f+1)3d + 2Cd & \text{for} & n \geq 2f+1 \\ (3\frac{f}{n-f} + 5)(f+1)3d + 2Cd & \text{for} & n \leq 2f. \end{array}$$

The bound of Section 3.5.2 can be slightly modified to give a bound of

$$(2\sqrt{2C} + 6)(f+1)3d + 2Cd \quad \text{for} \quad n \leq 2f.$$

Using the second timeout task, in which a process waits for $d/(\mu c_1)$ steps between every pair of messages, adds a delay of up to $C(d/\mu)$ to every message. Each message is then in effect be delayed by a total of up to $d(1 + C/\mu)$. This gives bounds of

$$4(f+1)(1 + C/\mu)d + (C^2 d/\mu + Cd) \quad \text{for} \quad n \geq 2f+1$$

and

$$\left.\begin{array}{r} (3\frac{f}{n-f} + 5)(f+1)(1 + C/\mu)d + (C^2 d/\mu + Cd) \\ \text{and } (2C/\sqrt{\mu} + 2\sqrt{C} + 6)(f+1)d + (C^2 d/\mu + Cd) \end{array}\right. \quad \text{for} \quad n \leq 2f.$$

It may be possible that a more clever stategy would allow processes to send messages of the algorithm on demand by more closely intertwining the main algorithm and the fault-detection task.

Chapter 6

Conclusions

We first summarize the known bounds for consensus:

Failure type	$n \geq$	Lower bound	Upper bound	Reference
Stopping	$f+1$	$(f-1)d + Cd$, $(f+1)d$	$2(f+1)d + Cd$	[ADLS90]
Omissions (sending)	$2f+1$	$\uparrow$	$4(f+1)d + Cd$	Thm. 3.21
	$f+1$	$\uparrow$	$(3\frac{f}{n-f}+5)(f+1)d + Cd$, $(2\sqrt{C}+6)(f+1)d + Cd$	Thm. 3.25 Thm. 3.28
"Timing"	$f+1$	$\uparrow$	$(f+1)(Cd+d)$	(see below)
Byzantine	$3f+1$	$\uparrow$	$(f+1)(d+2Cd)$	Thm. 4.4
Auth. Byzantine	$2f+1$	$\uparrow$	$(f+1)(d+2Cd)$	(see below)
	$f+1$	$\left(\left\lceil\frac{f}{n-f}\right\rceil + \left\lfloor\frac{f}{n-f}\right\rfloor\right) Cd$	$(C^{f+1} + C^f + \cdots + C)d$	(see below)

The bounds for stopping and omission failures (for $n \geq 2f+1$) are tight to within approximately a constant factor (2 and 4, respectively). The bounds for omission failures when $n \leq 2f$ are not tight; an improvement in either direction would be interesting. It has been noted by Bharali ([B91]) that the running time for omissions failures can be improved to $3(f+1)d + Cd$ by the following modification to the algorithm. The improvement is obtained by reducing the delay caused by a process that must wait for acknowledgments before

[0]This is an updated version of the original Chapter 6. It differs by the inclusion of the following: the upper bound for authenticated Byzantine failures when $n \geq 2f+1$, the improvement of the constant from 4 to 3 for the running time of the omissions algorithm ([B91]), and the more careful analysis of the running time in the model of [HK89].

sending an r message. Recall that if p_i sends an $r-1$ message at exactly t_{r-1}—the latest possible time—and immediately thereafter receives an r message, it may have to wait until time $t_{r-1}+2d$ to receive enough acknowledgments for its $r-1$ message before sending an r message and advancing to phase $r+1$. Thus a process p_j receiving the r message from p_i would not receive it until $t_{r-1}+3d$. The idea is to let p_i send a "virtual r message", even though it has not yet received $n-f$ acknowledgments for its $r-1$ message. Process p_j does not treat a virtual r message from p_i as a regular r message until it sees that p_i has received enough acknowledgments for its $r-1$ message (recall that all messages, including acknowledgments are broadcast to all processes). Thus, if p_i does get enough acknowledgments, then both p_i and p_j receive them by time $t_{r-1}+2d$ and p_j has effectively received a (real) r message from p_i by time $t_{r-1}+2d$, saving time d.

For failures less benign than omissions, this thesis leaves open a large gap in time complexity. In particular, the following central question remains unanswered:

> Does consensus in the presence of Byzantine failures require time $\Omega(fCd)$?

The difficulty of this problem seems to lie not in the potential of for arbitrary message content but in the potential for timing misbehavior. We believe an important step towards answering this question will be to obtain tight bounds for "timing failures", described below.

Timing failures

We say that a process suffers a "timing failure" if the time between some pair of successive steps is not in the interval $[c_1, c_2]$. The simple direct rounds simulation first described in Chapter 3 tolerates timing failures as well, implying a consensus algorithm with running time $(f+1)(Cd+d)$. The algorithm of [ADLS90] is also correct despite timing failures, but each of its phases may take up to time $Cd+d$. In fact, no algorithm is known to tolerate timing failures in less than time $O(fCd)$.

Byzantine failures with authentication

First note that the direct simulations outlined at the beginning of Chapter 3 do not work in the presence of Byzantine failures, even with authentication, and our general simulation of Chapter 4 itself requires $n \geq 3f+1$.

The upper bound for authenticated Byzantine failures with $n \geq 2f+1$ can be obtained by a very simple modification of the algorithm for Byzantine failures: change "**If** $|V^r| \geq 2f+1$" to "relay V^{r+1}" (unconditionally). In other words, to ensure that every process receives $f+1$ round r messages (and therefore sends its own round r message), it is sufficient to relay the

$f+1$ round r messages already received—these messages are signed and therefore believable. This protocol works for $n \geq 2f+1$ and achieves the same time complexity.

When $n \leq 2f$, the only obvious algorithm to tolerate authenticated Byzantine failures is a costly simulation of a synchronous algorithm. The simulation requires that processses begin synchronized and time out each other's timeouts by terminating round i after $(C^{i-1} + \cdots + C + 1)d/c_1$ steps.

The lower bound for authenticated Byzantine failures, not presented in this thesis, is interesting (greater than Cd) only for the limited range of $n \leq 2f$, and therefore says nothing interesting about unauthenticated Byzantine failures. The proof of this bound is similar to the "shifting scenarios" proofs of [FLM86].

Before suggesting other directions for further research, we first comment on the implications of our bounds for consensus in a closely related model.

6.1 Consensus in the related model of [HK89]

Herzberg and Kutten [HK89] consider a model in which the actual worst-case message delay in a given execution, δ, may be much less than the a priori worst-case bound on message delay, Δ. It is thus desirable for the running time of algorithms to depend minimally on Δ. This model raises similar concerns as our model does; in particular, detecting the absence of a message may be much more expensive than receiving the message.

For the consensus problem, it is not difficult to see that direct implementation of synchronous algorithms gives a running time of $O(f\Delta)$ for any type of failures; on the other hand, clearly the synchronous lower bound implies that no algorithm can guarantee a running time of less than $(f+1)\delta$. In this model, our algorithms yield an improvement over direct simulation strategies similar to the corresponding improvement in our semi-synchronous model.

It is not difficult to see that our algorithms may be run without modification in the model of [HK89], yielding the same running times with the syntactic substitution of Δ for Cd and δ for d. Thus we obtain bounds of

$$\begin{array}{rl} 4(f+1)\delta + \Delta & \text{for} \quad n \geq 2f+1 \\ (3\frac{f}{n-f} + 5)(f+1)\delta + \Delta & \text{for} \quad n \leq 2f. \end{array}$$

The bound involving $\sqrt{C}$ carries over with $\sqrt{\Delta/\delta}$ instead of $\sqrt{C}$, giving a bound of $(2\sqrt{\Delta/\delta}+6)(f+1)\delta + \Delta$. If $\delta \ll \Delta$, these are significant improvements over the bounds obtainable by direct simulation. Moreover, it is possible to prove better bound for these algorithms in general, depending on the ratio of Δ to δ; the bound of $4(f+1)d + Cd$ is realized by an execution only if $\Delta = 4\delta$. This is because process clocks are perfectly synchronized: whereas in our model the time between failure and detection may be anywhere in the range $[d, d + Cd]$, in this model it must be Δ (plus or minus twice the step time, which is assumed to be much less than δ; see [ADLS90], §7).) The length of each phase except the last must therefore be Δ and must have at least $\Delta/\delta - 3$ failures (processes in B_r). There are thus at most $(f+1)/(\Delta/\delta - 3)$ phases except for the last phase, and the running time is at most $(\frac{\Delta}{\Delta - 3\delta})(f+1)\delta + \Delta$. The running time is the maximum of this expression and $4(f+1)\delta + \Delta$. Similarly for the stopping failures algorithm of [ADLS90], the running time is the maximum of $2(f+1)d) + Cd$ and $(\frac{\Delta}{\Delta - \delta})(f+1)\delta + \Delta$. Our algorithm for Byzantine failures is not interesting in this model, as it is trivial to design an algorithm taking only time $(f+1)\Delta$ (our algorithm takes $(2f+1)\Delta + f\delta$).

In comparison, the algorithms of [DLS88] may also be used in the model of [HK89]. For stopping and omission failures (sending and receiving), their algorithms require $n \geq 2f + 1$; for Byzantine failures, they require $n \geq 3f + 1$. Their algorithms assume only that an upper bound on message delay time *exists*—it may not be known to the processes; the running time is a function of the maximum message delay in the given execution. The running times are $O(\delta^2 + n^2)$ for all types failures. (As noted in [DLS88], the running times can be improved to $O(\delta^2 + f^2)$. We also note that the different model considered there, which enables processes to send to at most one process per step, does not affect the time bound asymptotically.)

Note that in contrast to our algorithm and the algorithm of [ADLS90], the running times of the algorithms of [DLS88] in the [HK89] model do not depend at all on Δ, the upper bound on message delay time. This is possible because the model of [HK89] provides an extra degree of power to the algorithm by assuming that process clocks are perfectly synchronized. The algorithms of [DLS88] do not give good bounds in our model; the running times depend only polynomially on the ratio of process step rate, C. This difference in the model also accounts for the simplicity of solving consensus in the presence of Byzantine failures in the [HK89] model, relative to our semi-synchronous model.

6.2 Directions for further research

There are many possible directions for interesting research addressing the issues and concerns of real-time behavior of distributed systems:

- The existence of the underlying synchronous algorithm described in Section 3.2 suggests that the results of [ADLS90] and this thesis may be generalizable to certain classes of synchronous algorithms. For instance, the properties of the underlying synchronous algorithm that make it amenable to "efficient" simulation in our model are that it is "early-stopping" and that processes advance to further rounds only because of messages received (not because of messages omitted).

  - Can the properties sufficient for efficient simulation be clearly characterized?
  - Can these properties be shown necessary by proving lower bounds with large dependency on C for synchronous algorithms lacking these properties?
  - Are the factors of $\frac{f}{n-f}$ and $\sqrt{C}$ inherent to such simulations with $n \leq 2f$ and tolerating omission failures?

- What classes of problems are in fact affected by timing uncertainty? Perhaps problems solvable in asynchronous systems need not be affected. Can they be helped by timing assumptions? Are only fault-tolerant problems affected?

- Similar questions can be asked in the context of the model of Herzberg and Kutten ([HK89]): What can be said about converting synchronous algorithms with running times as a function of message delay d to algorithms that depend on the actual worst-case message delay δ rather than the a priori worst-case message delay Δ?

- What can be said about simulating synchronous algorithms that do not operate in rounds?

- Other work ([SDC90]) on the real-time complexity of the consensus problem assumes a different model of semi-synchrony. There, continuous local clocks are assumed to be within a fixed constant ϵ of each other and to stay within a linear envelope of real time. Insight into how these two models are related would enable a comparison of the bounds that have been obtained. In particular, using the assumptions of our model, for what values of their parameters can their model be implementable?

- We have given a straightforward implementation of our consensus algorithm using bounded capacity message links. Can a more involved approach avoid merely effectively increasing the delay of each message?

- For other problems, can bounded capacity message links be used to control implicitly message complexity by causing message inefficiency to be manifested as time inefficiency?

Bibliography

[ADLS90] 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.

[AL89] 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.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM,* 32(4):804–823, October 1985.

[AM90] H. Attiya and M. Mavronicolas. Efficiency of asynchronous vs. semi-synchronous networks. *the 28th annual Allerton Conference on Communication, Control and Computing,* October 1990.

[B91] A. Bharali. Personal communication, August 1991.

[CASD86] F. Cristian, H. Aghili, R. Strong and D. Dolev. Atomic broadcast: from simple message diffusion to byzantine agreement. *Proc. 15th Int. Conf. on Fault Tolerant Computing,* 1985, pp. 1–7. Also: IBM Research Report RJ5244, revised October 1989.

[CD86] B. A. Coan and C. Dwork. Simultaneity is harder than agreement. *Proc. 5th IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1986, pp. 141–150.

[CT90] B. Coan and G. Thomas. Agreeing on a leader in real-time. *Proc. 11th IEEE Real-Time Systems Symposium,* 1990.

[DLM82] R. DeMillo, N. A. Lynch and M. Merritt. Cryptographic protocols. *Proc. 14th Annual ACM Symp. on Theory of Computing,* May 1982, pp. 383–400.

[DDS87] D. Dolev, C. Dwork and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM,* Vol. 34, No. 1 (January 1987), pp. 77–97.

[DRS82] D. Dolev, R. Reischuk, and H. R. Strong. Eventual is earlier than immediate. *Proceedings of the 23rd IEEE Symp. on Foundations of Computer Science*, 1982, pp. 196–203.

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing,* Vol. 12, No. 3 (November 1983), pp. 656–666.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, Vol. 35 (1988), pp. 288–323.

[DM86] C. Dwork and Y. Moses. Knowledge and common knowledge in byzantine environments I: crash failures. *Proc. 1st Conf. on Theoretical Aspects of Reasoning About Knowledge*, Morgan-Kaufmann, Los Altos, CA, 1986, pp. 149–170; *Information and Computation*, to appear.

[FL82] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters,* Vol. 14, No. 4 (June 1982), pp. 183–186.

[FLM86] M. Fischer, N. Lynch and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing,* No. 1, 1986, pp. 26–39.

[FLP85] 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.

[Gal82] R. G. Gallager. Distributed minimum hop algorithms. Technical report LIDS–P–1175, MIT, January 1982.

[H84] V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations,* Ph.D. Thesis, Harvard University, June 1984. Technical Report TR–11–84, Department of Computer Science, Harvard University.

[HK89] A. Herzberg and S. Kutten. Efficient Detection of Message Forwarding Faults. *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 1989, pp. 339–353.

[LSP82] L. Lamport, R. Shostak and M. Pease. The byzantine generals problem. *ACM Transaction on Prog. Lang. and Sys.,* Vol. 4, No. 3 (July 1982), pp. 382–401.

[LL84] 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.

[LA90] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Proceedings of the 9th ACM PODC*, pp. 265–280, August 1990. Expanded version appears as MIT/LCS/TM–412.b, December 1989. Submitted for publication.

[LG89] N. Lynch and K. Goldman. Distributed algorithms—lecture notes for 6.852 Fall 1989. MIT/LCS/RSS 5, May 1989.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *Proceedings of the 6th ACM PODC*, pp. 137–151, August 1987. Also, expanded version as MIT/LCS/TR–387, April 1987.

[LF81] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1): 17–43, January 1981.

[M85] M. Merritt. Notes on the Dolev-Strong lower bound for byzantine agreement. Unpublished manuscript, 1985.

[MMT90] M. Merritt, F. Modugno and M. Tuttle. Time constrained automata. Unpublished manuscript, August 1990.

[PSL80] M. Pease, R. Shostak and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM,* Vol. 27, No. 2 (1980), pp. 228–234.

[PF77] G. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. *Proceedings of the 9th STOC*, pp. 91–97, May 1977.

[SWL86] B. Simons, J. L. Welch and N. Lynch. An overview of clock synchronization. *Proceedings of IBM Fault-Tolerant Computing Workshop,* March, 1986.

[SDC90] R. Strong, D. Dolev and F. Cristian. New latency bounds for atomic broadcast. *11th IEEE Real-Time Systems Symposium*, 1990.

[WL88] J. L. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation,* Vol. 77, No. 1 (April 1988), pp. 1–36.