Transaction Commit in a Realistic Timing Model

Brian A. Coan
Bell Communications Research

Jennifer Lundelius Welch
Massachusetts Institute of Technology

Abstract: An important problem in the construction of fault-tolerant distributed database systems is the design of nonblocking transaction commit protocols. This problem has been extensively studied for synchronous systems (i.e., systems where no messages ever arrive late). In this paper, the synchrony assumption is relaxed. A new partially synchronous timing model is given. In this model, a new nonblocking randomized transaction commit protocol is given, based on a Byzantine agreement protocol of Ben-Or. The new protocol works as long as fewer than half the processors fail. A lower bound is proved, showing that the number of processor faults tolerated is optimal. The protocol exhibits a graceful degradation property: when more than half the processors fail, the protocol blocks, but no processor produces a wrong answer. A notion of asynchronous round is defined and the protocol is shown to terminate in a small constant expected number of asynchronous rounds. The final result is that no protocol in this model can terminate in a bounded expected number of steps, even if processors are synchronous.

Keywords: Distributed databases, transaction commit, time bounds, fault tolerance, lower bounds, randomized protocols.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, by the National Science Foundation under Grant DCR-83-02391, by the Office of Army Research under Contract DAAG29-84-K-0058, and by the Office of Naval Research under Contract N00014-85-K-0168. A preliminary version of this paper appeared in the *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Systems* [CL].

1. Introduction

In a distributed database system a transaction may be processed concurrently at several different processors. To maintain the integrity of the database these processors must take consistent action regarding the transaction — either the results of the transaction are installed in the database at all processors (the transaction is *committed*), or the results are installed at no processor (the transaction is *aborted*). The decision whether to abort or commit a transaction is made by a *transaction commit protocol.* The objective for such a protocol is to commit as many transactions as possible subject to the constraint that each processor must be able to abort a transaction unilaterally.

A transaction commit protocol must never produce inconsistent decisions, and it must allow unilateral aborts. It has some leeway, though. Some protocols can produce more aborts than others, and some protocols fail to terminate in some situations. If failures can cause some nonfaulty processors to remain undecided about the fate of a transaction (at least as long as the failure persists), a processor is said to *block*, and the protocol is called *blocking.* Otherwise, the protocol is *nonblocking.* The most common transaction commit protocol in practice, two phase commit, is a blocking protocol. A blocking protocol is preferable in real systems to one that allows inconsistent decisions to be made, since it allows consistent decisions to be reached after the failures are repaired. A nonblocking protocol would be more preferable still.

Many elegant nonblocking transaction commit protocols [S] [DS] have been developed for completely synchronous systems. An obstacle to using these protocols in real systems is that a single violation of the timing assumptions (i.e., a late message) can cause the protocol to produce the wrong answer. The most common alternative timing model, the completely asynchronous model, unfortunately does not allow any solution to the transaction commit problem, either randomized or deterministic.[1] We give a new timing model that is intermediate between the synchronous and asynchronous models previously studied. In this model, we give a new nonblocking transaction commit protocol.

---

[1]The intuition behind this impossibility result is the following. Suppose there is a protocol that works in an asynchronous system, and guarantees that nonfaulty processors eventually decide (with probability 1); if the processors all begin with commit and there are no failures, then they all decide commit; and if any processor begins with abort, then the nonfaulty processors decide abort. Consider a run in which all processors but p begin with commit and are nonfaulty, while p fails initially. Eventually, the rest of the processors must decide. Since p could have started with abort, the processors must decide abort. But

We model real systems in which messages are usually delivered within some known time bound but sometimes come late. We do this by assuming a completely asynchronous system, in which relative processor speeds are unbounded and messages can take arbitrarily long to arrive, and letting the timing behavior affect the correctness conditions for the transaction commit problem, as follows. If every processor initially wants to commit the transaction, then the common decision must be to commit, provided no processors fail and all messages arrive within some known fixed time bound. If any processor initially wants to abort the transaction, then the common decision must be to abort, no matter what the timing and fault behavior of the system is. This problem definition takes advantage of the leeway allowed in specifying when processors must commit. Assuming that failures and late messages are relatively rare, the overall progress of the transaction processing system will not be impeded very much. A similar division is made in [DLS], in which properties that must always hold are separated from properties that only need hold when the system is well-behaved. In most other respects our model differs from theirs.

We prove that in our model no transaction commit protocol can terminate in a bounded expected number of steps. Consequently a new measure is needed to analyze the time performance of our protocol. One of the contributions of this paper is such a measure, which we call an asynchronous round. Our definition of asynchronous round is strong enough to allow us to show that our protocol terminates in a small constant expected number of asynchronous rounds. In Section 2 we argue that this notion of asynchronous round is not unrealistically strong.

Randomization is needed in the protocol because a result of [DDS] implies that no deterministic protocol is possible. In order to analyze a randomized protocol, we must define the adversaries against which the protocol will work. Our notion of the adversary is drawn from [CMS]. The adversary in our model chooses the order in which processors take steps, when each message will be delivered, and which processors fail and when (as long as fewer than half fail). It makes these decisions dynamically, during the execution of the protocol, using unlimited computational power. The adversary has available at any point in the execution all information about the hardware and software of the processors, and the pattern of communication up to that time, but it does not know the contents of the messages sent, nor the local states of processors, nor the processors' local random choices, unless that

---

there is another run that looks identical up to the decision point to all the processors except p, in which p begins with commit, and all its messages are delayed until after the decision is made. But in this run, the decision should have been commit.

information is deducible from the pattern of communication. We will be careful to design our protocol so that it is not deducible.

Our protocol uses a modified version of a solution to the agreement problem. In the *agreement problem* each processor begins with an initial value, 0 or 1, and decides on a final value. All nonfaulty processors' final values must be equal, and if all processors have the same initial value, then that value must be the final value. Thus if one processor begins with 0 and the rest with 1, either 0 or 1 is a correct answer to the agreement problem, whereas in the transaction commit problem, the answer must be 0 (if 0 is identified with abort).

An important difference between the transaction commit problem and the agreement problem is that in the former, all processors that decide are required to agree, including processors that decide and subsequently fail. This strict agreement condition is imposed because we assume that failed processors will eventually recover. The hope is that processors that fail and subsequently recover can be reintegrated using a separate recovery protocol. Skeen's thesis has an excellent discussion of recovery protocols [S]. We do not discuss these protocols further in this paper.

We assume that the faulty processors fail by crashing (i.e., stopping without warning). This is a realistic assumption that is commonly made in the database literature [S]. The number of faults tolerated by our protocol is optimal, since we prove a matching lower bound. Our protocol works as long as more than half the processors are nonfaulty. An important property of our protocol is that it degrades gracefully if the bound on the number of faulty processors is exceeded — instead of producing a wrong answer, the protocol simply fails to terminate.

At the beginning of our protocol, processors exchange some messages, and then execute a modification of Ben-Or's asynchronous agreement protocol [Be] to decide the fate of the transaction. The preliminary message exchanges serve two purposes: first, the differences between the input-output relations for the transaction commit and agreement problems are resolved, and second, a number of identical random bits are distributed.[1] These identical random bits are used in the agreement protocol to lower the expected running time from exponential to constant. There is a body of work dealing with attaining constant expected running time for the agreement

[1] We have not solved the global coin toss problem, however, because our protocol does not guarantee that the identical random bits are successfully distributed; the nature of the transaction commit problem, as discussed above, is such that our protocol can tolerate this failure.

problem [R] [CMS]; our technique does not solve this problem, for the following reason. In our protocol, if the identical random bits are not distributed in a timely fashion, processors can unilaterally decide 0 (abort), because we are solving the transaction commit problem. Such action is not an option for processors trying to solve the agreement problem, because it could violate the condition that all processors decide 1 if they all start with 1.

The transaction commit protocols of Skeen [S] and Dwork and Skeen [DS] tolerate any number of processor faults, while our protocol only handles fewer than half of the processors failing. However, if half or more of the processors fail, our protocol does not produce a wrong answer but merely fails to terminate, leaving open the opportunity for processors to recover. Late messages are not a problem for our protocol because of our model, but as we noted earlier they can cause the protocols in [S] and [DS] to produce a wrong answer.

In summary, the principal contributions of this paper are a realistic timing model, a method for analyzing the time performance of protocols in this model, an efficient fault-tolerant protocol for the transaction commit problem, and lower bounds showing that the protocol has optimal fault-tolerance, and that no protocol can terminate in a constant expected number of steps for each processor.

Following an exposition of our formal model in Section 2, we present our randomized transaction commit protocol in Section 3. Section 4 contains the lower bound proof showing that our protocol tolerates the maximal number of faulty processors. Finally, in Section 5 we show that no transaction commit protocol can guarantee that each processor terminate in a bounded expected number of its own steps, even if processors are synchronous.

2. Model

Processors are modeled as state machines that communicate by sending messages. Messages can take arbitrarily long to arrive. Our protocol works even in a very weak model in which there is no bound on the relative frequency with which processors take steps, and in which there is no atomic broadcast of messages. Our lower bound results are shown for the stronger case in which processors run in lockstep synchrony and possess atomic broadcast. In this section we present the weaker model. In Sections 4 and 5 we indicate the necessary changes for the stronger model. Our model is similar to those in [FLP] and [DDS].

Throughout this paper, 1 is identified with "commit" and 0 with "abort."

2.1 Basic Model

A *raw message* consists of some text, and the names of the sending and receiving processors. A *message* is a (raw message, integer) ordered pair; the integer denotes the sending time, as will be explained later. The reason for distinguishing between messages and raw messages is that we do not wish to require timestamps on all (raw) messages sent by processors, yet this information is useful in the exposition of the model for distinguishing multiple instances of the same raw message and determining message delays.

A *processor* is an infinite state machine, together with a message buffer, and a random number generator. The message buffer holds messages that have been sent to the processor but not yet received, and is modeled as a set of messages. The random number generator supplies an infinite sequence of n-bit strings. The state machine's transition function uses the current state, current random bit string and set of raw messages received to compute the new state and raw messages to be sent. Certain states are initial states, designated $(id, initval)$, where id is a nonnegative integer and $initval$ is either 0 or 1. The id element of the initial state is the processor's name, or identification number. The $initval$ element is the processor's initial value. Each processor can send zero or one message to every processor in one step. There is an integer in each processor's state, called its *clock*, which is 0 in an initial state, and is always incremented by 1 by the transition function. Thus, the clock counts how many steps the processor has taken so far. A *protocol* is a set of n processors.

A *configuration* C consists of n states, one for each processor, and n sets of messages, one for each processor's buffer. An *initial configuration* has all processors in initial states and all buffers equal to the empty set.

An *event* is denoted (p, M, b), in which processor p receives the set of messages M (which can be empty), and the random bit string b.

An event $e = (p, M, b)$ is *applicable* to configuration C if every message in M is an element of p's buffer in C. Let s and M' be the state and set of raw messages resulting from applying p's transition function to p's state in C, b, and the raw messages extracted from M. The configuration resulting from applying e to C, denoted $e(C)$, is obtained from C by removing all messages in M from p's buffer, changing p's state to s, and adding the message (m, i), for each $m \in M'$, to the appropriate buffer, where i is the value of p's clock in s.

A *schedule* is a finite or infinite sequence of events. A finite schedule $\sigma = e_1 e_2 \ldots e_k$ is *applicable* to configuration C if e_1 is applicable to C, e_2 is applicable to $e_1(C)$, etc. The resulting configuration is denoted $\sigma(C)$. An infinite schedule is applicable to C if every finite prefix of the schedule is applicable to C.

Given configuration C_1 and schedule σ applicable to C_1, we define the *run* $R = run(C_1, \sigma)$ obtained from C_1 and σ, as follows. If $\sigma = e_1 e_2 \ldots e_k$ is finite, then R is the sequence $C_1 e_1 C_2 e_2 \ldots e_k C_{k+1}$, where $C_{i+1} = e_i(C_i)$, $1 \leq i \leq k$. If $\sigma = e_1 e_2 \ldots$ is infinite, then R is the sequence $C_1 e_1 C_2 e_2 \ldots$, where, for all i, $C_1 e_1 C_2 e_2 \ldots e_i C_{i+1} = run(C_1, e_1 e_2 \ldots e_i)$. We also denote σ by $sched(R)$. Informally, a run is a schedule together with its associated configurations.

Processor p is *nonfaulty* in an infinite run or schedule if it takes an infinite number of steps; otherwise it is *faulty*. An infinite run or schedule is *failure-free* if no processor is faulty in it. Since the interleaving of processors' steps in a run or schedule may be arbitrary, no particular degree of synchronization is necessarily achieved.

A message sent by processor p at event e in infinite run R is *guaranteed* if e is not the last step of p in R. An infinite run R is *t-admissible*, for $0 \leq t \leq n$, if

- the first configuration is an initial configuration,
- at most t processors are faulty, and
- all guaranteed messages sent to nonfaulty processors are eventually received.

The notion of guaranteed messages is used to model the lack of atomic broadcast. Since messages sent at a processor's last step do not have to be received, we effectively model a processor failing in the middle of a broadcast.

There are two disjoint sets of *decision states*, Y_0 and Y_1, such that if a processor enters a state in Y_0 or Y_1 it stays in that set forever. A processor *decides* v when it is in a state in Y_v. A run is *deciding* if every nonfaulty processor decides. A configuration C has *decision value* v if there is some processor whose state in C is an element of Y_v.

2.2 Timing Constraints

We fix a positive constant $K \geq 1$, which is used to define late messages. A message m from p to q is *late* in run $R = C_1 e_1 C_2 e_2 \ldots$ if event e_s adds m to q's message buffer, and one of the following is true. (1) There is no event in R that removes m from q's message buffer, and some processor takes more than K steps in R after e_s. (2) There is an event e_r that removes m from q's message buffer,

and some processor takes more than K steps in the schedule $e_{s+1} \ldots e_r$. A run is *on-time* if it contains no late messages.

Ideally we would like a processor to decide in a constant expected number of its own steps. Unfortunately, as we prove in Section 5, we cannot do this, even if processors run in lockstep synchrony. Instead, we characterize the time performance of our protocol using the following definition. Given an infinite run, a processor is defined inductively to be in a particular *asynchronous round* (or *round*) as follows. Asynchronous round 1 begins for processor p when p first takes a step and ends after p's K^{th} step. Asynchronous round r, $r > 1$, begins for p at the end of p's round $r-1$ and ends either K of p's steps after the end of p's round $r-1$, or as soon as p receives every received message sent by a processor q in q's round $r-1$, whichever happens later. (We say "every *received* message" in order to make sure that no round lasts infinitely long due to p's waiting for a non-guaranteed message from q that never arrives.)

This definition uses two criteria for ending a round, the number of processor steps taken and the collection of messages received. These criteria seem natural in our timing model, in which processors can take actions depending on the receipt of messages, as well as on timeouts.

A processor cannot compute its current asynchronous round; the definition is for our use as omniscient observers as we analyze protocols. The reason we require a round to last at least K steps is to prevent a round from collapsing to nothing if no messages are sent in the previous round. If processors take steps in round-robin order, and receive and send messages only at the beginning of a round, and if each message sent at the sender's i^{th} step is received at the recipient's $(i+K)^{th}$ step (for all i), then this definition is essentially the same as the synchronous round definition in [DS]. Thus this definition is not unreasonably strong.

2.3 Safety Conditions

The following definition restricts what must happen if a processor decides, but does not require any processor to decide. A protocol is a *transaction commit protocol* if for every t-admissible run R:

- *Agreement Condition:* Every configuration has at most one decision value.
- *Abort Validity Condition:* If the initial value of any processor is 0, then no configuration has decision value 1.

- *Commit Validity Condition:* If the initial value of all processors is 1 and R is failure-free and on-time, then no configuration has decision value 0.

To exclude uninteresting protocols, we require that each processor must be able to receive at least n messages at each step. Otherwise, processors could swamp the message system, causing messages to become late not because the message system misbehaves, but because the ability of the processors to handle all the incoming message traffic is inadequate.[1] For instance, the protocol "cause the run to be not on-time by flooding the message system and then abort" is not of much practical interest.

2.4 Adversary

The adversary can be considered a scheduler — it decides which processor takes a step next and what messages are received. In the introduction we gave an informal description of the adversary. This subsection formalizes the notion.

The *message pattern* of finite run $R = C_1 e_1 \ldots e_k C_{k+1}$, where $e_i = (p_i, M_i, f_i)$ for all $1 \leq i \leq k$, is the sequence of triples $(p_1, E_1, P_1) \ldots (p_k, E_k, P_k)$, where P_i is the set of processors to which messages were sent by event e_i, and E_i is a set of integers indexing the events in the run that sent the messages, M_i, received in e_i. The point of making this definition is to isolate the pattern of message sending and receiving while hiding the contents of the messages.

An *adversary* is a function that takes a message pattern, and returns a processor p and a set E of integers (which may be empty) satisfying the following condition. If i is in E, then in the i^{th} element of the message pattern, (p_i, E_i, P_i), p is in P_i (i.e., there actually was a message sent to p at the i^{th} event), and in no element of the message pattern does p receive this message (i.e., the message in question has not yet been received). Thus, the adversary decides on the next processor to take a step, plus a collection of messages to be received.

Let $\mathcal{F}$ be the collection of all n-tuples of infinite sequences of n-bit strings. Each element of $\mathcal{F}$ is a possible set of choices returned by the n processors' random number

---

[1] Suppose each processor can send n messages per step but only receive $n-1$. Consider the protocol: At each step, broadcast a message; at step 1, decide 0. We now show that no infinite run is on-time. Let R be an infinite run. After $Kn(n-1)+n$ events, $(Kn(n-1)+n)n$ messages have been sent, and at most $(Kn(n-1)+n)(n-1)$ have been received. So there are at least $Kn(n-1)+n$ outstanding messages. By the pigeonhole principle, some processor p has at least $K(n-1)+1$ outstanding messages (to be received). It will take p at least $K+1$ steps to receive all those messages, by which time the run will no longer be on-time.

generators in an infinite run. A run is uniquely determined by an adversary A, an initial configuration I, and an element F of $\mathcal{F}$. Denote this run by $run(A, I, F)$. The construction of $run(A, I, F) = C_1 e_1 C_2 e_2 \ldots$ is inductive. Let $C_1 = I$. Suppose the run up to configuration C_i has been constructed. Let p and E be the result of A acting on the message pattern of run $C_1 e_1 \ldots C_i$. Then e_i consists of the processor p, the messages sent to p in all the events indexed by E, and the next unused bit string in the sequence for p in F. Finally, $C_{i+1} = e_i(C_i)$. Since the adversary is a total function, $run(A, I, F)$ is an infinite run, and thus at least one processor is nonfaulty.

If the adversary were not restricted in any way, it could cause all processors (but one) to fail or no messages to be delivered, and no protocol would be possible. We limit the power of the adversary in the following reasonable way. We define a *t-admissible* adversary to be an adversary such that for all initial configurations I and all F in $\mathcal{F}$, $run(A, I, F)$ is t-admissible.

For predicate P defined on runs, let $\Pr[P]$ be the probability of the event $\{F \in \mathcal{F} : run(A, I, F) \text{ satisfies } P\}$, for a fixed adversary A and initial configuration I.

The expected value of any complexity measure for a fixed randomized protocol is defined as follows. Let T be a random variable that given a run returns the complexity measure of interest for that run. For fixed t-admissible adversary A and initial configuration I, let the expected value of T, taken over the random numbers F, be denoted $E(T_{A,I})$. Define the expected value for the protocol, ET, to be $\max_{A,I}\{E(T_{A,I})\}$.

2.5 Liveness Condition

Given infinite run R and integer r, let $\text{DONE}(R, r)$ be the predicate that every nonfaulty processor decides by its asynchronous round r in R. A protocol is *t-nonblocking* if for any t-admissible adversary A and any initial configuration I,

$$\lim_{r \to \infty} \Pr[\text{DONE}(run(A, I, F), r)] = 1.$$

2.6 Problem Statement

Our goal is to design a t-nonblocking transaction commit protocol.

3. The Randomized Commit Protocol

Our protocol to solve the transaction commit problem is based on the asynchronous agreement protocol in [Be]. Similar protocols have been widely used [Br] [CC] [CMS] [R]. For the rest of this section, we assume a fixed t with $n > 2t$.

3.1 The Protocol

In this subsection, we present the randomized transaction commit protocol by describing, for each processor p, the states and transition function of p. First, we give an informal description.

Throughout the protocol each processor keeps a *vote* telling what it currently wants to do with the transaction. The processor with *id* 0 is the *coordinator*; at its first step, it chooses n random bits and distributes them to the other processors, the *participants*, by broadcasting a coins message containing the bits. If a participant receives no message at its first step, it sends a request message to the coordinator (to try to jog it awake); if no reply is received within 2K steps, the participant sets its vote to 0 and decides 0. If a participant receives a message at its first step, it extracts the n bits and broadcasts them in a coins message, to indicate "I am participating in the protocol." If a processor does not receive a coins message from everyone within 2K steps after broadcasting one, it sets its vote to 0 and decides 0. Then each processor broadcasts its vote. If a processor does not receive n votes for 1 within a short time, it sets its vote to 0, but remains undecided.

The rest of the protocol proceeds in stages (as in [Be]), numbered from 1 up without bound. In stage s, each processor p broadcasts its vote in a stage $(s,1)$ message and waits to receive $n-t$ stage $(s,1)$ messages. If p receives more than $n/2$ stage $(s,1)$ messages with vote $v \in \{0,1\}$, then p broadcasts v in a stage $(s,2)$ message; otherwise p broadcasts "?" in a stage $(s,2)$ message. Then p waits to receive $n-t$ stage $(s,2)$ messages. If p receives a stage $(s,2)$ message with value $v \in \{0,1\}$, then p sets its vote to v; otherwise, p sets its vote to a random bit, either the s^{th} random bit from the coins message if $s \leq n$, or else a locally-determined random bit. If p receives $n-t$ stage $(s,2)$ messages for value $v \in \{0,1\}$, then p decides v.

Processor p uses the following constants, variables and subroutines. Constants are p, n and K. Variables are:

- $clock_p$: nonnegative integer; initially 0.

- $stage_p$: values are "asleep," "request," "coins," "vote," $(s, 1)$ and $(s, 2)$ for all $s \geq 1$; initially "asleep."
- $timer_p$: nonnegative integer or *nil*; initially *nil*.
- $coins_p$: n-bit string or *nil*; initially *nil*.
- $vote_p$: boolean; initially p's initial value.
- $decide_p$: boolean or *nil*; initially *nil*.

The text of each raw message consists of the sending processor's current stage, and optionally a *value* (0, 1 or "?"), and an n-bit string.

Below we describe p's transition function, acting on state q of p, set M of raw messages, and n-bit string b. The description consists of several clusters of pseudocode. Each cluster is preceded by a predicate on q and M. The predicate of at most one cluster is true for any q and M. The state of p returned by the transition function is obtained from q by incrementing $clock_p$ by 1, remembering the set M, and then executing the cluster (if any) whose predicate is true of q and M. The set of raw messages returned by the transition function is that indicated by the send and broadcast statements of the appropriate cluster. If no cluster is true, then no raw messages are sent, the only changes to the state are that $clock_p$ is incremented and the received messages are remembered.

/* coordinator initiates protocol by distributing n random bits */

$stage_p$ = "asleep" for p = coordinator:
  $coins_p := b$
  $stage_p :=$ "coins"
  $timer_p := clock_p + 2K$
  broadcast $(stage_p$,"?",$coins_p)$

/* non-coordinator wakes up and requests that coordinator initiate */

$stage_p$ = "asleep" for $p \neq$ coordinator and $M = \emptyset$:
  $stage_p :=$ "request"
  $timer_p := clock_p + 2K$
  send "request" to coordinator

/* non-coordinator receives coins */

$stage_p$ = "asleep" or "request" for $p \neq$ coordinator and
there is a message in M with text $(s, v, coins)$:
  $coins_p := coins$

$stage_p$:= "coins"
$timer_p := clock_p + 2K$
broadcast ($stage_p$,"?",$coins_p$)

/* non-coordinator times out while waiting to receive coins */

$stage_p$ = "request" and $clock_p = timer_p$:
vote$_p$:= 0
decide$_p$:= 0

/* distributing votes */

$stage_p$ = "coins" and either $clock_p = timer_p$ or n coins messages have been received:
$stage_p$:= "vote"
$timer_p := clock_p + 2K$
if less than n coins messages have been received then [
$vote_p$:= 0
$decide_p$:= 0]
broadcast ($stage_p$,$vote_p$,$coins_p$)

/* completing stage 0 */

$stage_p$ = "vote" and either $clock_p = timer_p$ or n vote messages have been received:
$stage_p$:= (1,1)
if n votes for 1 have been received
then $vote_p$:= 1
else $vote_p$:= 0
broadcast ($stage_p$, $vote_p$, $coins_p$)

/* finishing first part of stage s */

$stage_p = (s, 1)$ and at least $n - t$ stage $(s, 1)$ messages have been received:
$stage_p := (s, 2)$
if more than $n/2$ stage $(s, 1)$ messages received have value v, for some v,
then broadcast ($stage_p$, v, $coins_p$)
else broadcast ($stage_p$,"?", $coins_p$)

/* finishing second part of stage s */

$stage_p = (s, 2)$ and at least $n - t$ stage $(s, 2)$ messages have been received:

$stage_p := (s+1, 1)$
if a stage $(s,2)$ message received has value v, for some v, then [
$vote_p := v$
if at least $n-t$ stage $(s,2)$ messages received have value v then $decide_p := v$
]
else if $s \leq n$ then $vote_p := coins_p[s]$ else $vote_p :=$ first bit of b
broadcast $(stage_p, vote_p, coins_p)$

Transaction Commit Protocol: p's transition function on input M, b, and arbitrary state

---

3.2 Proof of Correctness

The proof is organized as follows. Section 3.2.1 shows the safety properties, i.e., that the protocol is a transaction commit protocol. Section 3.2.2 contains the probabilistic analysis, which is applied to show the t-nonblocking property in Section 3.2.3.

3.2.1 Safety Conditions

Section 3.2.1 culminates in Theorem 8, which shows that the protocol is a transaction commit protocol.

All the lemmas in Section 3.2.1 hold for any (infinite) run from an initial configuration. In particular, they hold for runs in which more than t processors fail. Stating these results in this way allows us to show the graceful degradation property of the protocol.

In run R, processor p is said to be *in stage* s, for $s \geq 1$, if $stage_p = (s,1)$ or $(s,2)$. We say p *completes* stage $s \geq 0$ if p ever sets $stage_p$ to $(s+1,1)$ in R. Let p's decision states Y_0 and Y_1 be states with $decide_p = 0$ and $decide_p = 1$ respectively; Lemma 7 below shows that once p enters a state in Y_v, it stays in that set forever. Note that if no nonfaulty processor ever receives a coins message, then no processor completes stage 0.

Lemma 1: *In any run from an initial configuration, if some processor p has $vote_p = 0$ initially, then every stage (1,1) message has value 0.*

Proof: No processor ever receives a vote message with value 1 from p. Thus no processor sets its vote to 1 at the end of its vote stage, and no processor broadcasts a stage (1,1) message with value 1. □

Lemma 2: *In any infinite run from an initial configuration, if every processor p has $vote_p = 1$ initially, and the run is failure-free and on-time, then every processor broadcasts a stage (1,1) message with value 1.*

Proof: First we show that each processor p broadcasts a vote message with value 1. Suppose either p is the coordinator, or p receives a message at its first step. Then p broadcasts a coins message at its first step. By time K on p's clock, each processor receives p's coins message and broadcasts its own coins message (if it has not already done so). By time $2K$ on p's clock, p receives n coins messages. Thus p broadcasts a vote message with value 1.

Now suppose p is not the coordinator and does not receive any messages at its first step. It sends a request message to the coordinator, which is received by time K on p's clock. The coordinator then broadcasts a coins message, if it has not already done so, and p receives the coins message by time $2K$ on p's clock. Then p broadcasts a coins message; by time $3K$ on p's clock, each processor receives p's coins message and broadcasts its own coins message (if it has not already done so). By time $4K$ on p's clock p receives n coins messages. Thus p broadcasts a vote message with value 1.

Now we show that every processor p receives n vote messages within $2K$ of its clock ticks after it broadcasts its vote. Processor p broadcasts its vote as soon as it receives its n^{th} coins message. Suppose its clock reads T then. Since the run is on-time, every other processor receives its n^{th} coins message, and broadcasts its vote, by the time p's clock reads $T + K$. Thus p receives all n vote messages by the time its clock reads $T + 2K$. Then p broadcasts its stage (1,1) message with value 1. □

Lemma 3: *In any run from an initial configuration, if every stage (s,1) message has value $v \in \{0,1\}$, then every processor that completes stage s decides v at stage s, for any $s \geq 1$.*

Proof: Let p be any processor that broadcasts a stage $(s,2)$ message. Then p receives at least $n-t$ stage $(s,1)$ messages, all with value $v \in \{0,1\}$ by assumption. Since $n > 2t$, $n - t > n/2$. Thus p broadcasts a stage $(s,2)$ message with value v.

Now let p be any processor that completes stage s. Then p receives at least $n-t$ stage $(s,2)$ messages, all with value v. Thus p decides v. □

For any $s \geq 1$, we call a stage $(s,2)$ message with value $v \in \{0,1\}$ an *S-message* ("S" for "set"), because the receipt of such a message can cause a processor to set

its vote to v (if this message is among the first $n-t$ stage $(s,2)$ messages received by the processor).

Lemma 4: *In any run from an initial configuration, there is at most one value sent in S-messages during any stage $s \geq 1$.*

Proof: In order to send an S-message for some value v at stage s, a processor must receive more than $n/2$ stage $(s,1)$ messages with value v. Since processors do not broadcast conflicting messages, fewer than $n/2$ processors can broadcast a stage $(s,1)$ message with value $w \neq v$. Thus, no processor receives more than $n/2$ stage $(s,1)$ messages with value w, and no processor sends an S-message for w at stage s. □

Lemma 5: *In any run from an initial configuration, if any processor decides v before stage 1, then*
(1) $v = 0$, and
(2) every processor that completes stage 1 decides v by the end of stage 1.

Proof: Suppose p decides before stage 1.

(1) By inspecting the code, we see that p decides 0, and sets its vote to 0 before broadcasting its vote message.

(2) As in the proof of Lemma 1, every stage (1,1) message has value 0, and by Lemma 3, every processor that completes stage 1 decides 0. □

Lemma 6: *In any run from an initial configuration, if some processor decides v at stage $s \geq 1$, then*
(1) no processor decides $w \neq v$ at stage s, and
(2) every processor that completes stage $s+1$ decides v at stage $s+1$.

Proof: Suppose processor p decides v at stage $s \geq 1$. Let q be any processor that completes stage s. Since p decides v at stage s, it receives at least $n-t$ stage $(s,2)$ messages with value v before completing stage s. Thus, since $n > 2t$ and q receives at least $n-t$ stage $(s,2)$ messages before completing stage s, at least one of these messages is from a processor from which p receives an S-message for v in stage s. Since processors do not broadcast conflicting messages, q receives at least one S-message for v at stage s. By Lemma 4, q sets its vote to v, and thus q broadcasts a stage $(s+1,1)$ message with value v.

(1) If q decides in stage s, then q decides v.

(2) By Lemma 3, every processor that completes stage $s+1$ decides v at stage $s+1$. □

Lemma 7: *In any run from an initial configuration, $decide_p$ changes value at most once, for every processor p.*

Proof: Pick any processor p. If $decide_p$ is set before stage 1, then by Lemma 5, every processor that completes stage 1 decides v at stage 1. If $decide_p$ is set for the first time in stage $s \geq 1$, then by Lemma 6, every processor that completes stage $s+1$ decides v by the end of stage $s+1$. Lemma 3 shows that for any $r \geq 1$, if every processor that completes stage r decides v at stage r, then any processor that completes stage $r+1$ decides v at stage $r+1$.

Theorem 8: *Protocol 1 is a transaction commit protocol.*

Proof: Let R be a t-admissible run. First we show the agreement condition, that there is at most one decision value in every configuration of R. If some processor decides before stage 1, then Lemmas 5 and 7 give the result. If no processor decides until stage $s \geq 1$, then Lemmas 6 and 7 give the result.

Next we show the abort validity condition. Suppose some processor begins with initial value 0. If no processor completes stage 0, then Lemma 5 shows that no processor decides 1. If some processor completes stage 0, then all nonfaulty processors complete stage s, for all $s \geq 0$. Lemmas 1 and 3 (with $u = t$) give the result.

Finally, we show the commit validity condition. Suppose R is failure-free and on-time, and all processors begin with 1. Then Lemmas 2 and 3 give the result. □

Since Lemmas 1 through 7 are true for any (infinite) run from an initial configuration, the agreement, abort validity, and commit validity conditions are true even for runs in which more than t processors fail. This is the graceful degradation property exhibited by our protocol.

3.2.2 Probabilistic Properties

The analysis in this subsection is directed toward showing that the probability that all processors that complete stage s, decide by stage s, apporaches 1 as s increases. Recall that probabilities are taken over the random information, holding the adversary and initial configuration fixed.

For the following definitions, fix adversary A, initial configuration I, and F and F' in $\mathcal{F}$. Let $R = run(A, I, F)$ and $R' = run(A, I, F')$.

Define $F(p, k)$ to be the k^{th} element in the sequence for p in F.

Define $coins(F)$ to be $F(0, 1)$ (i.e., the coordinator's first n-bit string). It is easy to see that if $coins_p$ is ever nonnil in R, then it equals $coins(F)$, for all p. We denote the s^{th} element of $coins(F)$ by $coins(F)[s]$.

For processor p and $s \geq 1$, define $index(R, p, s)$ to be the number of steps taken by p to complete stage s in R. If p does not complete stage s, then $index(R, p, s)$ is undefined. Thus $index(R, p, s)$ is also the index into the sequence for p in F of the bit string used to determine the value of $vote_p$ in stage s, in case $s > n$ and p receives no S-message in stage s.

The next definition maps a bit to each processor and each stage $s > n$ in a run, such that each stage gets "new" bits. This mapping is consistent with the mapping implemented in the protocol for those cases where a processor uses a random bit. Let $random(R, p, s)$, for processor p and $s > n$, be defined as follows. (1) If p completes stage s in R, then $random(R, p, s)$ is the first bit of $F(p, k)$, where $k = index(R, p, s)$. (2) If p does not complete stage s in R, then $random(R, p, s)$ is the second bit of $F(p, s + 1)$ (i.e., a safe default).

For $0 \leq s \leq n$, define F and F' to be (A, I, s)-*equal* if $coins(F)[i] = coins(F')[i]$ for all i, $1 \leq i \leq s$. For $s > n$, define F and F' to be (A, I, s)-*equal* if F and F' are (A, I, n)-equal, and for every i, $n + 1 \leq i \leq s$, and every processor p, $random(R, p, s) = random(R', p, s)$.

For $s \geq 1$, define $v(R, s)$ to be the value of an S-message sent in run R at stage s. If no S-message is sent in R at stage s, then let $v(R, s) = 0$. By Lemma 4, $v(R, s)$ is uniquely defined.

Define MATCH(R, s) to be the predicate that if $s \leq n$, then $coins(F)[s] = v(R, s)$, and if $s > n$, then $random(R, p, s) = v(R, s)$ for all p.

Define SAME(R, s) to be the predicate that all processors that complete stage s in R set their votes to the same value in stage s.

Define DECIDE(R, s) to be the predicate that each processor that completes stage s has decided by the end of stage s in R.

The next lemma characterizes two aspects of runs that are unchanged, once an adversary and initial configuration are fixed.

Lemma 9: *Let A be an adversary, I an initial configuration, and F and $F' \in \mathcal{F}$. Let $R = run(A, I, F) = C_1 e_1 C_2 \ldots$ and $R' = run(A, I, F') = C'_1 e'_1 C'_2 \ldots$.*
(1) For all $i \geq 1$, the message pattern of $C_1 e_1 \ldots C_i$ is the same as the message pattern of $C'_1 e'_1 \ldots C'_i$.
(2) For all processors p and all $s \geq 1$, $index(R, p, s) = index(R', p, s)$.

Proof: (1) The structure of the protocol is such that the random information does not affect which processors send messages to which other processes — it only affects the values of the local variables and the message contents. But this is the very information not available to the adversaries under consideration. Thus, for a fixed adversary and initial configuration, the sequence of processor steps and the message delays are the same, regardless of the random information.

(2) Follows from (1). □

The next lemma states that the value of an S-message sent in stage $s+1$ only depends on the random information available through stage s, once an adversary and initial configuration are fixed.

Lemma 10: *Let $R = run(A, I, F)$ and $R' = run(A, I, F')$ for adversary A, initial configuration I, and F and F' in $\mathcal{F}$. If F and F' are (A, I, s)-equal, then $v(R, s+1) = v(R', s+1)$, for any $s \geq 0$.*

Proof: By Lemma 9, the message patterns for R and R' are the same. Since F and F' are (A, I, s)-equal, the random information that affects the local variables and message contents in R and R' up through stage s is the same in F and F'. Thus, the values of corresponding processors' variables, and the contents of corresponding messages sent up through stage s are the same in R and R'. The random information used in stage $s+1$ is not used until the end of stage $s+1$, so the same messages are sent in stage $s+1$ in R and R', even though the stage $s+1$ random information might be different in F and F'. □

The next lemma states some simple relationships between MATCH, SAME, and DECIDE.

Lemma 11: *Let $R = run(A, I, F)$ for adversary A, initial configuration I and $F \in \mathcal{F}$. For all $s \geq 1$,*
(1) MATCH(R, s) implies SAME(R, s), and

(2) SAME(R, s) *implies* DECIDE$(R, s+1)$.

Proof: Fix $s \geq 1$.

(1) If $s \leq n$, then MATCH(R, s) means that $coins(F)[s] = v(R, s)$. Thus $coins_p$ has the same value as any S-message sent in stage s of R, for all p. Thus, each processor that completes stage s sets its vote to $v(R, s)$, and SAME(R, s) is true.

If $s > n$, then MATCH(R, s) means that the first bit of $F(p, k)$, where $k = index(R, p, s)$, is equal to the value of any S-message sent in stage s of R, for all p. Thus, each processor that completes stage s sets its vote to $v(R, s)$, and SAME(R, s) is true.

(2) If SAME(R, s) is true, then all stage $(s+1, 1)$ messages have the same value $v \in \{0, 1\}$. Thus all stage $(s+1, 2)$ messages have value v. Thus, every processor that completes stage $s+1$, decides v, and DECIDE$(R, s+1)$ is true. □

The following technical lemma concerns any equivalence class of $\mathcal{F}$, where the equivalence is defined by (A, I, s)-equality.

Lemma 12: *Fix adversary A, initial configuration I, and $s \geq 0$. Partition $\mathcal{F}$ into the maximal equivalence classes, within each of which all elements are (A, I, s)-equal. Pick any class C.*
(1) MATCH$(run(A, I, F), i) =$ MATCH$(run(A, I, F'), i)$ *for all i, $1 \leq i \leq s$, and any F and F' in C.*
(2) If $s < n$, then MATCH$(run(A, I, F), s+1)$ *is true for half the elements F of C; if $s \geq n$, then* MATCH$(run(A, I, F), s+1)$ *is true for a $1/2^n$ fraction of the elements F of C.*

Proof: (1) Choose any i, $1 \leq i \leq s$, and any F and F' in C. Let $R = run(A, I, F)$ and $R' = run(A, I, F')$. Since F and F' are $(A, I, i-1)$-equal, $v(R, i) = v(R', i)$, by Lemma 10. Since F and F' are (A, I, i)-equal, $coins(F)[i] = coins(F')[i]$ if $i \leq n$, and $random(R, p, i) = random(R', p, i)$ for all p if $i > n$; thus MATCH$(R, i) =$ MATCH(R', i).

(2) By Lemma 10, $v(run(A, I, F), s+1)$ is the same for all $F \in C$.

Suppose $s < n$. In half the elements F of C, $coins(F)[s+1] = 0$, and in half $coins(F)[s+1] = 1$, since all the elements of C are (A, I, s)-equal. Thus MATCH$(run(A, I, F), s+1)$is true for half the elements F of C.

Suppose $s \geq n$. Let $R = run(A, I, F)$ for F in C. MATCH$(R, s+1)$ means $random(R, p, s+1) = v(R, s+1)$ for all p. The position of $random(R, p, s+1)$ in F depends on whether p completes stage $s+1$ in R or not. By Lemma 9, either p completes stage $s+1$ in R for all F in C, or p fails to complete stage $s+1$ in R for all F in C. If p does not complete stage $s+1$, then $random(R, p, s+1)$ is the second bit of $F(p, s+2)$, obviously a fixed position for all F in C. If p does complete stage s, then $random(R, p, s)$ is the first bit of $F(p, k)$, where $k = index(R, p, s)$. By Lemma 9, k is the same for all F in C, so this is also a fixed position for all F in C. The positions of $random(R, p, s)$ for all p are all distinct. Thus a $1/2^n$ fraction of the elements F of C have $random(R, p, s) = v(R, s)$ for all p. □

The next lemma is the key to the termination of the protocol, as well as the good time performance. It says that there is a high probability that the random information used to set votes matches the value in S-messages for the first n stages, and there is a smaller, but still positive probability for subsequent stages.

Lemma 13: *Fix adversary A and initial configuration I. Then*

$Pr[\text{MATCH}(run(A, I, F), s)] = 1/2$ *if* $s \leq n$, *and* $1/2^n$ *if* $s > n$.

Proof: By part (2) of Lemma 12, since the lemma is true for every equivalence class of $\mathcal{F}$. □

The next lemma shows that the events of not matching in different stages are independent.

Lemma 14: *Fix adversary A and initial configuration I. Let $R = run(A, I, F)$ for $F \in \mathcal{F}$. Then for any $s \geq 1$,*

$$\Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,s)] = \Pr[\neg\text{MATCH}(R,1)] \cdots \Pr[\neg\text{MATCH}(R,s)].$$

Proof: Pick any i, $1 \leq i \leq s$. We will show that

$$\Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i)]$$

$$= \Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i-1)] \cdot \Pr[\neg\text{MATCH}(R,i)].$$

Let X be the set of all $F \in \mathcal{F}$ such that $\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i-1)$ is true, where $R = run(A, I, F)$. Partition $\mathcal{F}$ into equivalence classes based on

$(A, I, i-1)$-equality. If F is in X, and F and F' are $(A, I, i-1)$-equal, then F' is also in X, by part (1) of Lemma 12. Pick any equivalence class C that is a subset of X. Part (2) of Lemma 12 gives the result. □

The next lemma shows that the probability that all processors that complete stage s, decide by stage s, approaches 1 as s increases.

Lemma 15: *For any adversary A and initial configuration I,*

$$\lim_{s \to \infty} \Pr[\text{DECIDE}(run(A, I, F), s)] = 1.$$

Proof: Let $R = run(A, I, F)$. First note that

$$\Pr[\text{DECIDE}(R, s)] \geq \Pr[\text{MATCH}(R, 1) \vee \ldots \vee \text{MATCH}(R, s-1)].$$

The reason is that if $\text{MATCH}(R, s')$ is true for some s', $1 \leq s' \leq s-1$, then by Lemma 11, $\text{SAME}(R, s')$ is true, and thus $\text{DECIDE}(R, s'+1)$ is true. Since $s'+1 \leq s$, $\text{DECIDE}(R, s)$ is true.

$\Pr[\text{MATCH}(R, 1) \vee \ldots \vee \text{MATCH}(R, s-1)]$

$$\begin{aligned}
&= 1 - \Pr[\neg\text{MATCH}(R, 1) \wedge \ldots \wedge \neg\text{MATCH}(R, s-1)] \\
&= 1 - \prod_{i=1}^{s-1} (1 - \Pr[\text{MATCH}(R, i)]), \text{ by Lemma 14} \\
&\geq 1 - (1 - 1/2^n)^{s-1}, \text{ by Lemma 13.}
\end{aligned}$$

Since $\lim_{s \to \infty} (1 - 1/2^n)^{s-1} = 0$ we are done. □

3.2.3 Liveness Condition

Lemmas 16 and 17 convert Lemma 15 into a statement about the predicate DONE, in order to show the t-nonblocking property in Theorem 18.

Lemma 16: *In any run from an initial configuration, each processor that completes stage 0 without having decided is in at most asynchronous round 6.*

Proof: Suppose p completes stage 0 without having decided. Then p obtains the n random bits in some message by its $2K^{th}$ step, and broadcasts its coins message. At most $4K$ steps later, p completes stage 0. Since each asynchronous round lasts at least K steps, at most 6 rounds elapse. □

The next lemma shows that each stage $s \geq 1$ takes only a bounded number of asynchronous rounds.

Lemma 17: *In any run from an initial configuration, if each processor that completes stage $s \geq 0$ is in at most asynchronous round r when it completes stage s, then each processor that completes stage $s+1$ is in at most asynchronous round $r+2$ when it completes stage $s+1$.*

Proof: Let p be any processor that broadcasts a stage $(s+1,1)$ message. This happens when p completes stage s, so all stage $(s+1,1)$ messages are at most round r messages.

Let p be any processor that broadcasts a stage $(s+1,2)$ message. Processor p cannot enter round $r+1$ until it has received the last of the round r messages, including all the stage $(s+1,1)$ messages. Immediately after receiving the last of these (if not before), p broadcasts its stage $(s+1,2)$ message, so all stage $(s+1,2)$ messages are at most round $r+1$ messages.

No processor p can enter round $r+2$ until it has received the last of the round $r+1$ messages, including all the stage $(s+1,2)$ messages. Yet by the time p receives all the stage $(s+1,2)$ messages, p has completed stage $s+1$. □

Theorem 18: *Protocol 1 is t-nonblocking.*

Proof: Pick any t-admissible run R. Suppose no nonfaulty processor p receives a coins message in R. Then p decides 0 by time $2K$ on its clock, i.e., by round 2. Now suppose some nonfaulty processor receives a coins message in R. Then, since R is t-admissible, every nonfaulty processor receives a coins message in R, and completes stage s, for all $s \geq 0$. By Lemmas 16 and 17, $\text{DECIDE}(R,s)$ implies $\text{DONE}(R,6+2s)$ for any t-admissible run R. Lemma 15 gives the result. □

3.3 Time Complexity

Recall that expectation is defined in Section 2.4 to be taken over t-admissible adversaries and initial configurations. First, we show that the expected number of stages is less than 4.

Lemma 19: *Let X be a random variable giving the least s such that all processors that complete stage s decide by stage s. Then $EX < 4$.*

Proof: Fix t-admissible adversary A and initial configuration I. Let $R = run(A,I,F)$, for F in $\mathcal{F}$. Let $q_s = \Pr[\neg\text{MATCH}(R,s)]$. Let Y be a random variable

giving the least number s such that all processors that complete stage s have the same vote at the end of stage s. By Lemma 3, $X \leq Y + 1$.

$$
\begin{aligned}
EX \leq E(Y+1) &= 1 + EY = 1 + \sum_{s=1}^{\infty} s \cdot \Pr[Y = s] \\
&\leq 1 + \sum_{s=1}^{\infty} s \cdot \Pr\left[\left(\textstyle\bigwedge_{i=1}^{s-1} \neg \textsc{match}(R,i)\right) \wedge \textsc{match}(R,s)\right] \\
&= 1 + \sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_{s-1}(1 - q_s), \text{ by Lemma 14} \\
&= 1 + \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_{s-1}\right) - \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_s\right) \\
&= 1 + 1 + \left(\sum_{s=1}^{\infty} (s+1) \cdot q_1 q_2 \cdots q_s\right) - \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_s\right) \\
&= 2 + \sum_{s=1}^{\infty} (s + 1 - s) \cdot q_1 q_2 \cdots q_s \\
&= 2 + \sum_{s=1}^{\infty} q_1 q_2 \cdots q_s \\
&= 2 + \left(\sum_{s=1}^{n} q_1 \cdots q_s\right) + \left(q_1 \cdots q_n \cdot \sum_{s=n+1}^{\infty} q_{n+1} \cdots q_s\right).
\end{aligned}
$$

We simplify using specific values for q_s. For $1 \leq s \leq n$, $q_s = 1/2$, and for $s > n$, $q_s = 1 - 1/2^n$, by Lemma 13.

$$
\begin{aligned}
EX &\leq 2 + \sum_{s=1}^{n} \frac{1}{2^s} + \frac{1}{2^n} \cdot \sum_{s=n+1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s-n} \\
&< 2 + 1 + \frac{1}{2^n} \cdot \sum_{s=1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s} \\
&= 3 + \frac{1}{2^n}\left(\frac{1 - 1/2^n}{1 - (1 - 1/2^n)}\right) \\
&= 3 + \frac{1}{2^n}(2^n - 1) \\
&< 4.
\end{aligned}
$$

□

Theorem 20: *All nonfaulty processors decide in a constant expected number of asynchronous rounds.*

Proof: Let $R = run(A, I, F)$ for some t-admissible adversary A, initial configuration I, and $F \in \mathcal{F}$. If no nonfaulty processor receives a coins message in R, then every nonfaulty processor decides by round 2.

Suppose some nonfaulty processor receives a coins message in R. Then, since R is t-admissible, every nonfaulty processor p receives a coins message in R, and completes stage s, for all $s \geq 0$. By Lemma 16, p is in at most asynchronous round 6 when it completes stage 0. By Lemma 17, when p completes stage s of Protocol 1, it is in at most asynchronous round $6 + 2s$. The expected number of stages is 4, by Lemma 19. Therefore all nonfaulty processors decide in 14 expected asynchronous rounds. □

4. Lower Bound on Number of Processors

The lower bounds proved in the next two sections hold even if processors run in lockstep synchrony and possess an atomic broadcast capability. In this section, we first give relevant details of this stronger model, and then show that the number of faults tolerated by our transaction commit protocol is optimal.

A processor failure is represented by an explicit *failure step,* denoted $(p, \perp, b)$. After a failure step for p, p is in a distinguished *failed* state. Thus failures can be evidenced in finite runs. (Of course, processors cannot detect failures because message delivery is asynchronous.) A processor is *faulty* in a run if it takes a failure step, otherwise it is *nonfaulty.*

Processors take steps in round-robin order, 0 through $n-1$; a schedule of the form $(0, M_1, f_1) \ldots (n-1, M_n, f_n)$ is a *cycle.* To enforce the round-robin behavior, each configuration has a *turn* component, designating which processor's turn it is to take a step. An initial configuration has $turn = 0$. In order for an event $e = (p, *, b)$ to be applicable to a configuration C, $turn(C)$ must equal p, and if p is in the failed state in C, then e must be a failure step. After an event is applied, the resulting configuration's *turn* component is incremented by 1 (modulo n).

The *guarantee* definition is no longer needed, since atomic broadcast is allowed. The *delay* of message m that is received in run R is the number of the cycle to which the receiving event belongs minus the sending time of m. An infinite run R is *t-admissible*, for $0 \leq t \leq n$, if

- the first configuration is an initial configuration,
- at most t processors are faulty,
- all messages sent to a nonfaulty processor are received, and

- all received messages have delay at least 1.

In this model, the adversary cannot schedule when processors take steps, but can only determine when a processor fails and what the message delays are.

In this section we show that no protocol, even a randomized one, can solve the transaction commit problem unless more than half the processors are nonfaulty. The intuition behind the proof is similar to that for the coordinated attack problem (first posed in [G]; also analyzed in [HM]). We partition the processors into two groups, each of size at most t. Given a run that decides 1 (in which all processors begin with 1), we work backwards from the end of the run to the beginning, delaying messages between the two groups and showing that the resulting runs must still decide 1. Eventually we get a run in which no messages between the groups are received, yet the processors decide 1. This situation leads to a contradiction, since one group could have started with 0's, in which case the decision should be 0.

The actual construction of the runs is fairly involved, and is facilitated by the following definitions and lemmas.

Let $state(p, C)$ be the state of processor p in configuration C, and $buff(p, C)$ be the state of p's buffer in C. Given a schedule σ and a subset S of the processors, define $\sigma|S$ to be the subsequence of σ consisting of exactly those events that are steps for processors in S. Also define $kill(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, b)$ (where $*$ can be M or $\perp$) with $(p, \perp, b)$ whenever p is in S; similarly, define $deafen(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, b)$ (where $*$ can be M or $\perp$) with $(p, \emptyset, b)$ whenever p is in S.

Lemma 21: *Let σ be a schedule applicable to configuration C and τ be a schedule applicable to configuration D. Let S be a set of processors. If $state(p, C) = state(p, D)$ for all processors p in S and if $\sigma|S = \tau|S$, then for any processor p in S, $state(p, \sigma(C)) = state(p, \tau(D))$.*

Proof: Use induction on the length of $\sigma|S$, and the fact that the transition functions are deterministic, given states, messages and random numbers. □

Given a partition of the set of processors P into two sets S and S', define an *intergroup message* (relative to S and S') to be a message sent from a processor in S to a processor in S' or vice versa.

Lemma 22: *Let S and S' be a partition of the set of processors, and let C and D be two configurations such that $state(p, C) = state(p, D)$ and $buff(p, C) \subseteq buff(p, D)$*

for all p in S. Let σ be a schedule applicable to C in which any intergroup message that is received by $p \in S$ in σ is in $buff(p, C)$. Then

(a) the schedule $\phi = kill(S', \sigma)$ is applicable to D;

(b) if no processor in S' is in a failed state in D, then the schedule $\tau = deafen(S', \sigma)$ is applicable to D.

Proof: We show (b); (a) is similar. We proceed by induction on the length l of σ.

Basis: $l = 1$. Let $\sigma = e$ and $\tau = e'$. If e is an event for p in S', then p receives no messages in e'. This event is clearly applicable to D since p has not failed in D. If e is an event for p in S, then since $\tau = \sigma$ and $buff(p, C) \subseteq buff(p, D)$, the fact that σ is applicable to C implies that τ is applicable to D.

Induction: $l > 1$. Suppose the lemma is true for schedules of length $l - 1$ and show for length l. Let $\sigma = \sigma' e$ be a schedule of length l. Since σ' has length $l - 1$, by the induction hypothesis $\tau' = deafen(S', \sigma')$ is applicable to D. We must show that $e' = deafen(S', e)$ is applicable to $\tau'(D) = E$. If e is an event for p in S', then p receives no messages. This event is clearly applicable to E since p has not failed in D and no subsequent steps are failure steps.

Suppose $e = (p, M, b)$ for p in S. We must show that each m in M is in $buff(p, E)$. Choose m in M and let q be the sender.

If m is in $buff(p, C)$, then m is in p's buffer in every configuration from C to $\sigma'(C)$. Since $buff(p, C) \subseteq buff(p, D)$,and no message is removed from a buffer by τ' that is not removed by σ', m is still in $buff(p, E)$.

Suppose m is not in $buff(p, C)$. Then by assumption on σ, q is in S. Let $\sigma'' g$ be the prefix of σ' such that $(\sigma'' g)(C)$ is when m first appears in p's buffer. Thus, q sends m as a result of event g in $run(C, \sigma')$. Since q is in S, $\tau'' g$ is a prefix of τ', where $\tau'' = deafen(S', \sigma'')$. By the induction hypothesis, τ'' is applicable to D, so by Lemma 21, $state(q, \sigma''(C)) = state(q, \tau''(D))$. By the inductive hypothesis, since the length of $\sigma'' g$ is less than l, g is applicable to $\tau''(D)$. Thus m is also sent in $run(D, \tau')$, and m is in p's buffer in E. □

The next theorem shows that for any protocol, there is some finite run that computes the wrong decision value, if no more than half the processors are nonfaulty.

Theorem 23: *There is no t-nonblocking transaction commit protocol if $n \leq 2t$.*

Proof: Suppose $n \leq 2t$ and that there is a t-nonblocking transaction commit protocol with processors 0 through $n-1$.

Let $A = \{0, \ldots, t-1\}$ and $B = \{t, \ldots, n-1\}$. Each of A and B has at most t elements. The first t events of a cycle form an *A-semicycle* (each processor in A takes a step); the remaining events of a cycle form a *B-semicycle* (each processor in B takes a step). An infinite schedule applicable to an initial configuration consists of alternating A- and B-semicycles.

Let I_{11} be the initial configuration in which all processors have initial value 1. Since the protocol is a t-nonblocking transaction commit protocol, given an adversary that kills no processors and delivers in cycle $j+1$ any message sent in cycle j (so every run is failure-free and on-time), there is at least one finite deciding run $run(\alpha, I_{11})$ such that all processors have decided 1 in $\alpha(I_{11})$. Let $\alpha = \pi_1 \ldots \pi_y$ where each π_i is a semicycle.

Claim: There exist $y+1$ finite failure-free schedules α_1 through α_{y+1} such that for each i, (1) $\alpha_i = \pi_1 \ldots \pi_{i-1}\gamma_i$, (2) α_i is applicable to I_{11}, (3) all processors have decided 1 in $\alpha_i(I_{11})$, and (4) no intergroup message is received in γ_i.

Proof of Claim: We show the claim by descending induction on i. Let $C_i = (\pi_1 \ldots \pi_i)(I_{11})$ for $i \geq 1$, and $C_0 = I_{11}$.

Basis: $i = y+1$. Letting $\alpha_{y+1} = \alpha$ (so that γ_{y+1} is empty) proves the claim.

Induction: $i < y+1$. We assume the claim is true for $i+1$ and show it for i.

Assume π_i is a B-semicycle, i.e., i is even. (We will indicate in parentheses the changes necessary when π_i is an A-semicycle, i.e., when i is odd.) If no processor in B receives any message from a processor in A in π_i, then letting $\gamma_i = \pi_i\gamma_{i+1}$ satisfies properties (1) through (4).

Suppose some processor in B receives a message from some processor in A in π_i. We construct γ_i in two steps; first we construct β_1, after which all processors in A have decided, and then we construct β_2, in which all processors in B decide. Then γ_i will be $\beta_1\beta_2$.

Define β_1 to be $deafen(B, \pi_i\gamma_{i+1})$. (See Figure 1.) By Lemma 22, β_1 is applicable to C_{i-1}. Since $\beta_1|A = \pi_i\gamma_{i+1}|A$, Lemma 21 applies and each processor in A has the same state in $\beta_1(C_{i-1}) = F$ as it does in $(\pi_i\gamma_{i+1})(C_{i-1})$, so each decides 1 in F. No intergroup message is received in β_1 because processors in B receive no

messages in β_1, and processors in A receive no intergroup messages in $\pi_i\gamma_{i+1}$ or in β_1.

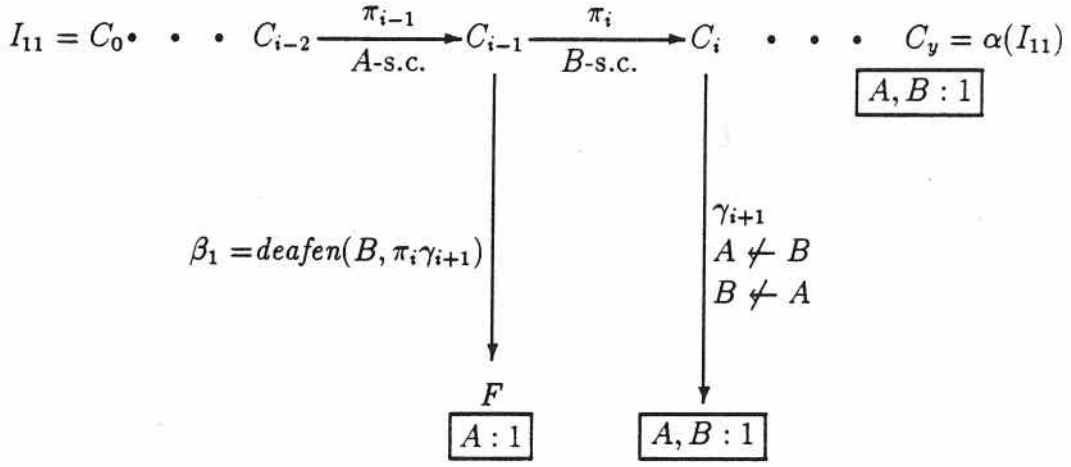

Figure 1: Construction of β_1

Now we must give a schedule β_2 that causes processors in B to decide 1 without hearing from any processors in A. The intuition is that processors in B must be able to decide without hearing from processors in A, because it is possible that all the processors in A have died. By the agreement condition, the processors in B must decide 1 also. The problem with applying this argument is that there may be leftover messages sent by processors in A before the point at which the processors in B think they died, and thus processors in B could wait to receive these messages before deciding. Thus, we must show that processors in A might have died even earlier.

Semicycle π_i is part of cycle number $\lceil i/2 \rceil = j$ in α_i. (See Figure 2.) Let D be the configuration in $run(\alpha_i, I_{11})$ immediately preceding the $(j-1)^{st}$ cycle of α_i. (If $j = 1$, then let $D = I_{11}$.) Let τ be the substring of α_i between I_{11} and D. Let ρ be the substring of α_i between D and C_{i-1}. There are two possibilities for ρ.

- If $i = 2$, then $D = I_{11}$ and $\rho = \pi_1$. Thus, ρ is an A-semicycle.
- If $i > 2$, then $D = C_{i-4}$ and $\rho = \pi_{i-3}\pi_{i-2}\pi_{i-1}$. Thus, ρ consists of all of cycle

$j-1$ and the first half of cycle j (an A-semicycle followed by a B-semicycle followed by another A-semicycle). (Pictured in Figure 2.)

(If π_i is an A-semicycle, i.e., if i is odd, then there are the following two possibilities for ρ.

- If $i = 1$, then $D = I_{11}$ and ρ is empty.
- If $i > 1$, then $D = C_{i-3}$ and $\rho = \pi_{i-2}\pi_{i-1}$. Thus, ρ consists of cycle $j-1$ (an A-semicycle, followed by a B-semicycle).)

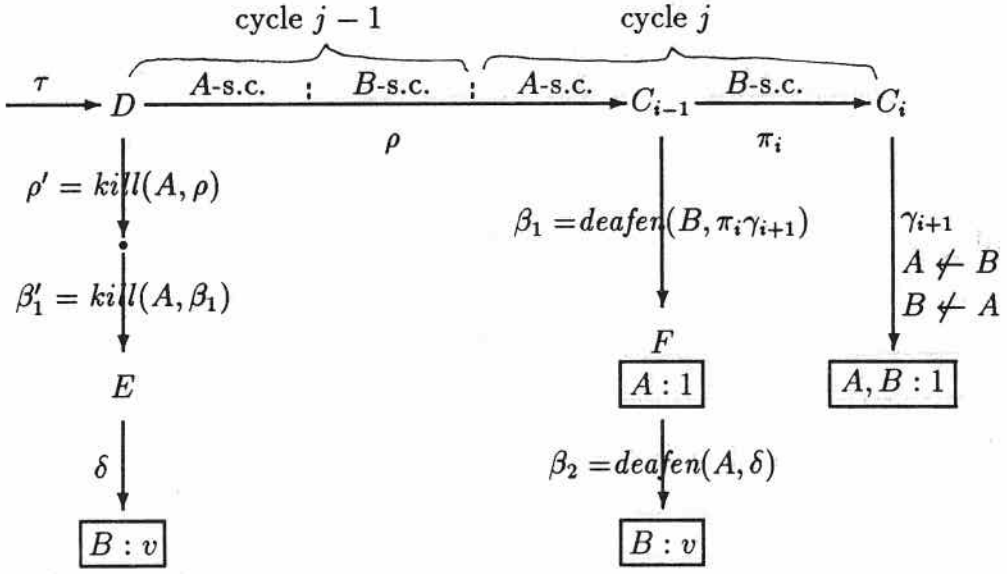

Figure 2: Construction of β_2

Let $\rho' = kill(A, \rho)$. Since no message is sent and received in the same cycle in α (and hence in ρ), any message received in ρ by a processor p in B from a processor in A is sent in $run(\tau, I_{11})$, i.e., prior to cycle $j-1$, and is in $buff(p, D)$. By Lemma 22, ρ' is applicable to D. Since $\rho|B = \rho'|B$, Lemma 21 implies that $state(p, \rho'(D)) = state(p, C_{i-1})$ for all p in B.

Consider the schedule $\beta_1' = kill(A, \beta_1)$. (See Figure 2.) Since the processors in A are failed and the processors in B receive no messages, β_1' is obviously applicable to $\rho'(D)$. Let $E = \beta_1'(\rho'(D))$. Since $\beta_1'|B = \beta_1|B$ and $state(p, \rho'(D)) = state(p, C_{i-1})$ for all p in B, Lemma 21 implies that $state(p, E) = state(p, F)$ for all p in B.

By the t-nonblocking property, since $|A| \leq t$, there must exist a finite deciding run from E with schedule δ. Suppose the decision value is v. Thus, all processors in B decide v in $\delta(E)$. By choice of α, all messages sent in $run(\tau, I_{11})$, i.e., before cycle $j-1$, are received by the end of cycle $j-1$, i.e., by the end of ρ or earlier. Since $\rho'|B = \rho|B$, every processor in B receives in ρ' all messages sent to it in $run(\tau, I_{11})$, i.e., before cycle $j-1$. Thus in δ, processors in B receive only messages sent in $run(\rho'\beta_1'\delta, \rho'(D))$. Since all processors in A are dead in $\rho'\beta_1'\delta$, B receives no intergroup messages in δ.

Let $\beta_2 = deafen(A, \delta)$. Pick p in B. From above, $state(p, E) = state(p, F)$. Let m be any message in $buff(p, E)$; m could only have been sent by a processor q in B in $run(\rho'\beta_1', D)$, i.e., in cycle $j-1$ or later. Lemma 21 implies that q has the same state in corresponding configurations in $run(\rho'\beta_1', D)$ and $run(\rho\beta_1, D)$. Thus q sends the same messages in the two runs, and m is also in $buff(p, F)$. Now we can apply Lemma 22 to show that β_2 is applicable to F.

Since $\beta_2|B = \delta|B$ and $state(p, F) = state(p, E)$ for all p in B, Lemma 21 implies that each processor p in B is in the same state in $\beta_2(F)$ as in $\delta(E)$. So each processor in B decides v in $\beta_2(F)$; by the agreement condition, $v = 1$, because processors in A have already decided 1 in F. No intergroup message is received in β_2 because none is received in δ.

Let $\gamma_i = \beta_1\beta_2$. We have shown that $\alpha_i = \pi_1 \ldots \pi_{i-1}\gamma_i$ satisfies properties (1), (2), (3) and (4). *End of Claim.*

Note that α_1 is a finite schedule in which no intergroup messages are received. Construct schedule $\sigma = kill(A, \alpha_1)$. By Lemma 22, σ is applicable to I_{11}. Since $\sigma|B = \alpha_1|B$, Lemma 21 implies that each processor in B has the same state in $\sigma(I_{11})$ as it does in $\alpha_1(I_{11})$, and thus also decides 1 in $\sigma(I_{11})$.

Let I_{01} be the initial configuration in which all processors in A have initial value 0 and all processors in B have initial value 1. By Lemma 22, σ is applicable to I_{01}. Since each processor in B begins with the same state in I_{01} as in I_{11}, by Lemma 21 each has the same state in $\sigma(I_{01})$ as it does in $\sigma(I_{11})$, and thus also decides 1 in $\sigma(I_{01})$. But this violates the abort validity condition. □

5. Lower Bound on Time

One might imagine a transaction commit protocol for our model such that each processor could decide in a constant number of its own steps, at least in many runs.

For instance, in the protocol presented in Section 3, at most $6K$ steps are required for a processor to complete stage 0 — a processor need not wait arbitrarily long for messages since the existence of a late message means that the processor is allowed to abort. Yet in the subsequent stages, no advantage is taken of this flexibility, and processors wait potentially unbounded time for messages. Unfortunately, the intuition that it may be possible to use the detection of late messages in order to shorten the running time (as measured in processor steps) is incorrect. In fact, in this section we prove that no protocol can guarantee that each processor terminate in a constant expected number of its own steps, even if processors run in lockstep synchrony, and even if only one processor can fail.

In particular, we show that for any constant B, there is a 1-admissible adversary and an initial configuration such that the expected number of cycles needed for all nonfaulty processors to decide is more than B. The proof is constructed as follows. We consider the initial configuration in which all processors begin with 1, and the adversary that kills no processors and delivers all messages with delay 1. If no run from this initial configuration with this adversary is deciding by cycle B, we are done. Suppose there is such a B-cycle run that is deciding. We find a point in this run that has the property there are some very long runs extending from this point that are not deciding. These runs are kept undeciding by delaying the delivery of all messages. These runs are so long that they cause the expected value to exceed B, when calculated with the appropriate initial configuration and adversary.

Thus, we must solve two subproblems. First, we must find the appropriate point in the run from which the long runs branch off (cf. Lemma 24); second, we must show that the long runs extending from this point are undeciding (cf. Lemma 25).

We need the following definitions in addition to the definitions and Lemmas 21 and 22 from Section 4.

If p is a processor, then schedule σ is *p-free* if p only takes failure steps in σ.

A run is *x-slow* for some constant x if every message received in the run has delay at least x. Given a configuration C, a schedule σ is *x-slow relative to C* if the run obtained by applying σ to C is x-slow.

A *seed* (for protocol P) is an n-tuple of sequences of n-bit strings, such that either each sequence is infinite or each sequence has the same number of elements. The *length* of a seed is the length of one sequence. If seed F has infinite length, then F is in $\mathcal{F}$. There is a finite number of seeds of any finite length.

A run is *F-compatible,* for seed F, if for all processors p and all i not exceeding the length of F, the random string that p receives in its i^{th} step is the same as the i^{th} element of p's sequence in F. Given configuration C, a schedule σ is *F-compatible relative to C* if C is reachable by an F-compatible run and $run(C, \sigma)$ is F-compatible.

For the remainder of this section, we fix an arbitrary 1-nonblocking transaction commit protocol P. From now on, "run" means a 1-admissible run of P, and "configuration" means a configuration reachable from some initial configuration of P by a 1-admissible run of P.

Let V be a subset of $\{0,1\}$, x an integer, and F a seed. Configuration C is *$\{x, F, V\}$-valent* if V is the set of decision values of all configurations that are reachable from C by an x-slow F-compatible run.

For the rest of this section, let I_1 be the initial configuration in which all processors have initial value 1.

The next lemma shows that in an F-compatible run that decides 1, there exists a configuration from which some F-compatible, x-slow run decides 1, and from which some other F-compatible, x-slow run decides 0.

Lemma 24: *If $run(I_1, \tau)$ is a finite failure-free on-time deciding run that is F-compatible for finite seed F, then for any integer $x > 0$ there exists a configuration in $run(I_1, \tau)$ that is $(x, F, \{0,1\})$-valent.*

Proof: Pick such a run $run(I_1, \tau)$ that is F-compatible, and fix x. By the commit validity condition, $\tau(I_1) = C$ has decision value 1. Thus all runs starting at C, including x-slow F-compatible runs, have decision value 1, and hence C is $(x, F, \{1\})$-valent.

Let I_{01} be the initial configuration in which some processor q has initial value 0 and the rest have initial value 1. Since the protocol is 1-nonblocking and since F is finite, there is a finite q-free x-slow F-compatible run $run(\sigma, I_{01})$ such that $\sigma(I_{01})$ has decision value 0, and by the agreement condition, $\sigma(I_{01})$ is $(x, F, \{0\})$-valent.

By Lemma 22, σ is also applicable to I_1. By Lemma 21, all processors except q have the same state in $\sigma(I_1)$ as in $\sigma(I_{01})$, and decide 0 in $\sigma(I_1)$. Thus I_1 is either $(x, F, \{0\})$-valent or $(x, F, \{0,1\})$-valent. If the latter is true, we are done. Suppose the former is true.

Since F is finite, by the 1-nonblocking property no configuration in $run(I_1, \tau)$ is $(x, F, \emptyset)$-valent. The valencies of I_1 and C imply that there must be an event $e = (p, M, b)$ and two adjacent configurations in $run(I_1, \tau)$, C_0 and C_1 with $C_1 = e(C_0)$, such that C_0 is either $(x, F, \{0\})$-valent or $(x, F, \{0, 1\})$-valent, and C_1 is either $(x, F, \{1\})$-valent or $(x, F, \{0, 1\})$-valent. (See Figure 3.)

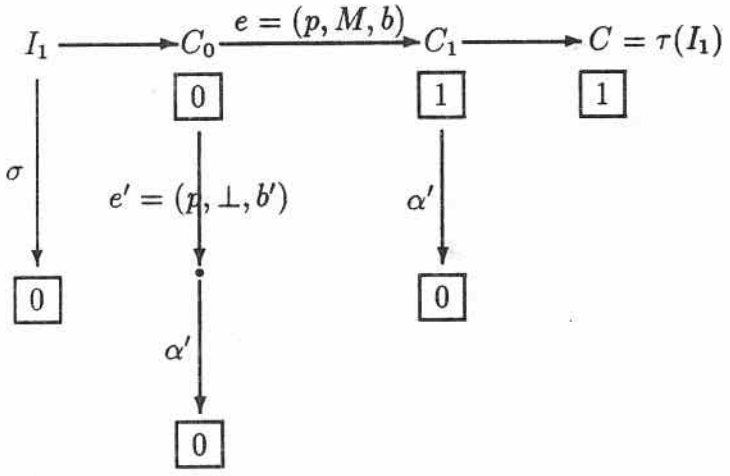

Figure 3: Demonstrating the existence of an $(x, F, \{0, 1\})$-valent configuration

If either configuration is $(x, F, \{0, 1\})$-valent, we are done. Say neither is. Since the protocol is 1-nonblocking, F is finite, no processor has failed so far, and C_0 is $(x, F, \{0\})$-valent, there is a finite p-free x-slow F-compatible run $run(\alpha, C_0)$ in which the nonfaulty processors decide 0. Say $\alpha = (p, \perp, b')\alpha'$. (If F is long enough to extend past C_0, then $b' = b$; otherwise, b' could differ from b.) Since α' is applicable to C_1, Lemma 21 implies that all the processors except p have the same state in $\alpha'(C_1)$ as they do in $\alpha(C_0)$. But since they decide 0 in $\alpha(C_0)$, and since α' is F-compatible and x-slow relative to C_1, this is a contradiction to the hypothesis that C_1 is $(x, F, \{1\})$-valent. □

The next lemma shows that in a certain situation, processors must remain undecided as long as no messages are received.

Lemma 25: *Let A be the adversary that kills no processors, and that for the first l events delivers messages after delay 1 and subsequently delivers messages after delay*

x, for some $x > l$. Let F be a seed of length x. If the configuration C following the l^{th} event in $run(A, I_1, F)$ is $(x, F, \{0,1\})$-valent, then the final configuration in $run(A, I_1, F)$ is $(x, F, \{0,1\})$-valent.

Proof: Let $run(A, I_1, F) = run(\alpha\sigma, I_1)$, where $C = \alpha(I_1)$. Assume in contradiction that $\sigma(C)$ is not $(x, F, \{0,1\})$-valent. Since F is finite, by the 1-nonblocking property, $\sigma(C)$ cannot be $(x, F, \emptyset)$-valent. Assume $\sigma(C)$ is $(x, F, \{v\})$-valent. Then there is a configuration D in $run(\sigma, C)$ and some event $e = (p, M, b)$ in σ such that D is $(x, F, \{0,1\})$-valent and $e(D)$ is $(x, F, \{w\})$-valent. M must be the empty set, since no messages are received in $run(\sigma, C)$. Suppose $w = 0$. (The argument is analogous if $w = 1$.) The only other event applicable to D that can be part of an x-slow F-compatible run is $(p, \perp, b) = e'$, because all messages sent more than x cycles ago have delay 1 and have already been received, and because F is long enough to extend to e. (See Figure 4.)

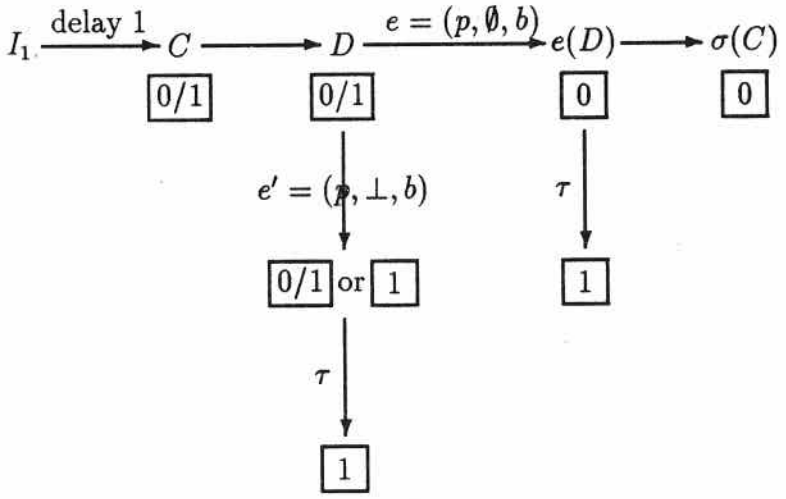

Figure 4: Demonstrating that $\sigma(C)$ is $(x, F, \{0,1\})$-valent

Since D is $(x, F, \{0,1\})$-valent, $e'(D)$ must be either $(x, F, \{0,1\})$-valent or $(x, F, \{1\})$-valent. Thus there is some finite p-free x-slow F-compatible run from $e'(D)$ that has decision value 1; let τ be its schedule. Now τ is also applicable, x-slow and F-compatible relative to $e(D)$, and all processors except p have the same state in $\tau(e(D))$ as in $\tau(e'(D))$ (by Lemma 21), so they decide 1, contradicting the valency of $e(D)$. □

Given infinite run R, let $T(R)$ be the cycle when the last nonfaulty processor decides.

Theorem 26: *For any constant B, there is a 1-admissible adversary A and an initial configuration I such that $E(T_{A,I}) \geq B$.*

Proof: Fix B. Let $\mathcal{R}$ be the set of all runs of the form $run(A_1, I_1, F)$, where F is a seed of length B, and A_1 is the adversary that kills no processors and delivers all messages with delay 1. Let $|\mathcal{R}| = j$. Thus, j is also the number of seeds of length B.

Case 1: No run in $\mathcal{R}$ is deciding. Let $A = A_1$ and $I = I_1$. Then $E(T_{A,I}) \geq B$.

Case 2: There is some run R in $\mathcal{R}$ that is deciding. Let $\mathcal{C}$ be the set of all configurations in run R, and let $m = |\mathcal{C}|$. Let $\mathcal{S}$ be the collection of all seeds with length jmB that extend the seed of R. $\mathcal{S}$ is finite; in fact, $|\mathcal{S}| = z/j$, where z is the total number of seeds of length jmB.

We will associate each seed in $\mathcal{S}$ with a configuration in $\mathcal{C}$ in such a way that all runs from a configuration in $\mathcal{C}$, using a particular adversary and any of the associated seeds, is undeciding. The extreme length of these undeciding runs will cause the desired expected value to exceed B.

For each $C \in \mathcal{C}$, define $S(C)$ to be the set of all $F \in \mathcal{S}$ such that C is the first $(jmB, F, \{0,1\})$-valent configuration in R. By Lemma 24, at least one $(jmB, F, \{0,1\})$-valent configuration exists in R; thus, each $F \in \mathcal{S}$ is in $S(C)$ for exactly one configuration C.

Fix C to be a configuration in $\mathcal{C}$ with $|S(C)| \geq \frac{1}{m} \cdot |\mathcal{S}|$. Such a configuration exists by the pigeonhole principle, since $|\mathcal{C}| = m$. Thus, $|S(C)| \geq \frac{1}{jm} \cdot z$.

Let l be the number of events that precede C in run R. Let A be the adversary that for the first l events delivers messages after delay 1 and that subsequently delivers messages after delay jmB. By Lemma 25, for every F in $S(C)$, the final configuration of $run(A, I_1, F)$ is $(jmB, F, \{0,1\})$-valent. Thus, no processor has decided in that final configuration, and $T(R') > jmB$, for any infinite run R' that is an extension of $run(A, I_1, F)$.

Let $I = I_1$. By choice of C, at least a $\frac{1}{jm}$ fraction of all the seeds of length jmB are in $S(C)$. Thus, at least a $\frac{1}{jm}$ fraction of all infinite seeds have a prefix in

$S(C)$. For any infinite seed F with a prefix in $S(C)$, $T(run(A, I, F)) > jmB$, by the argument above. As a result,

$$E(T_{A,I}) \geq \frac{1}{jm} \cdot jmB = B. \qquad \square$$

Acknowledgments

We thank Barbara Liskov, Nancy Lynch and Bill Weihl for suggesting this problem to us, Yoram Moses for helpful comments on an early draft, and Nancy Lynch for a very careful reading of a recent draft.

References

[Be] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," in *Proc. 2^{nd} Symposium on Principles of Distributed Computing,* pp. 27–30,1983.

[Br] G. Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Algorithm," *Proc. 17^{th} Ann. ACM Symp. on Theory of Computing,* pp. 316–326, 1985.

[CC] B. Chor and B. Coan, "A Simple and Efficient Randomized Byzantine Agreement Algorithm," *IEEE Trans. on Software Engineering,* vol. SE-11, no. 6, pp. 531–539, 1985.

[CL] B. Coan and J. Lundelius, "Transaction Commit in a Realistic Fault Model," *Proc. 5^{th} Ann. ACM Symp. on Principles of Distributed Computing,* pp. 40–51, 1986.

[CMS] B. Chor, M. Merritt, and D. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models," *Proc. 4^{th} Ann. ACM Symp. on Principles of Distributed Computing,* pp. 152–162, 1985.

[DDS] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM,* vol. 34, to appear.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Proc. 3^{rd} Ann. ACM Symp. on Principles of Distributed Computing,* pp. 103–118, 1984.

[DS] C. Dwork and D. Skeen, "The Inherent Cost of Nonblocking Commitment," *Proc. 2nd Ann. ACM Symp. on Principles of Distributed Computing,* pp. 1–11, 1983.

[FLP] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, 1985.

[G] J. Gray, "Notes on Data Base Operating Systems," Research Report RJ2188(300001)2/23/78, IBM Research Laboratory, San Jose, California, 1977.

[HM] J. Halpern and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. 3rd Ann. ACM Symp. on Principles of Distributed Computing*, pp. 50–61, 1984 (revised as of January 1986 as IBM-RJ-4421).

[R] M. Rabin, "Randomized Byzantine Generals," *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Science,* pp. 403–409, 1983.

[S] D. Skeen, "Crash Recovery in a Distributed Database System," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1982. (Also available as technical report UCB/BRL M82/45.)