Transaction Commit in a Realistic Fault Model

Brian A. Coan and **Jennifer Lundelius**
Massachusetts Institute of Technology

Abstract: We study the transaction commit problem under realistic timing assumptions. We identify an almost asynchronous model, which we claim is more realistic than some (synchronous) models that have been studied previously. In this model we give a randomized transaction commit protocol based on Ben-Or's randomized asynchronous Byzantine agreement protocol. The expected number of asynchronous rounds until our protocol terminates is a small constant, and the number of failstop faults tolerated is optimal. It is known that no deterministic protocol is possible in this model. We motivate our definition of asynchronous rounds by showing that no protocol in this model can terminate in a bounded expected number of clock ticks, even if processors are synchronous. Defining asynchronous rounds allows us to make the performance guarantee that after a sufficient number of useful messages have been delivered our protocol will terminate.

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*). Furthermore, each processor must be able to unilaterally abort the transaction. Ensuring that such consistent action is taken is the *transaction commit problem.*

Our transaction commit protocol works in an interesting new timing model that is intermediate between the synchronous and asynchronous models previously studied. We model real systems in which messages are usually delivered within some known time bound but sometimes come late. Many elegant transaction commit protocols [S] [DS] have been developed for the strictly synchronous model. The main difficulty in 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 model, the completely asynchronous model, unfortunately does not allow any solution to the transaction commit problem, either randomized or deterministic.

The way in which we model this partially synchronous system is to assume a completely asynchronous system, in which relative processor speeds are unbounded and messages can take arbitrarily long to arrive, and to let the timing behavior affect the correctness conditions. The transaction commit problem that we solve has the following correctness conditions. If every processor initially wants to commit the transaction, then the common decision must be to commit as long as 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 behavior of the system is. 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.

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. We assume that the faulty processors fail by crashing. The fail-stop assumption is realistic and is commonly made in the database literature [S].

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.

We prove that in our model no transaction commit protocol can terminate in a bounded expected amount of time. 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 show 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. To analyze our randomized protocol, we must define the adversary. 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. The adversary is limited to killing just under half the processors. 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 results of processors' local coin flips, unless that 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 solution to the agreement problem as a subroutine. 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).

Our agreement subroutine is a modification of Ben-Or's asynchronous agreement protocol [Be]. The modification lowers the expected running time from exponential to constant. A previous modification with the same purpose due to Rabin [R] requires a stronger model with a reliable distributor of coin flips. Chor, Merritt, and Shmoys [CMS] achieve the improved running time in a model that is stronger than Ben-Or's but more realistic than Rabin's. However their asynchronous protocol tolerates less than one-sixth of the processors failing. In the same model as [CMS] we improve on the fault tolerance of their protocol, while still achieving a constant expected running time, by supplying all processors with identical coin flips. A key part of our transaction commit protocol is an explicit strategy for distributing the identical coin flips. We believe that this strategy is not applicable to the problem solved in [CMS].

We compare our transaction commit protocol to those of Skeen [S] and Dwork and Skeen [DS]. Their protocols tolerate any number of processor faults. In contrast our protocol only handles less than half of the processors failing. However if the bound on the number of faults is exceeded, our protocol does not produce a wrong answer but merely fails to terminate. By not producing a wrong answer, we leave open the opportunity 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 our paper are a realistic partially synchronous 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 is optimal.

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 terminate in a bounded expected number of clock ticks as measured on any processor's clock, even if processors are synchronous.

2. Model

Processors are modeled as state machines that communicate by sending messages but without atomic broadcast. Messages can take arbitrarily long to arrive. There is no bound on the relative frequency with which processors take steps. Our protocol works even in a very weak model in which there is no bound on the relative frequency with which processors take steps. 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].

2.1 Basic Model

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 real numbers, distributed uniformly over the interval $[0,1)$. The state machine's transition function uses the current state, current random number and messages received to compute the new state and messages to be sent. Certain states are initial states, designated $(id, inital)$, where id is an integer and $inital$ is either 0 or 1. The id element of the initial state is the processor's identification number. The $inital$ element is the processor's initial value. Each processor can send up to one message to every processor in one step. There is an integer in each processor's state, called its *clock*, that 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, f), in which processor p receives the set of messages M (which can be empty), and the random number f. A processor must be able to receive at least n messages at a step (although it need not do so at every step, of course.)

An event $e = (p, M, f)$ is *applicable* to configuration C if every message in M is an element of p's buffer in C. 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 according to its transition function, and adding messages from p to the appropriate buffers according to the transition function. Processor p's transition function uses M, f, and p's state in C.

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.

We define the *run* R obtained from configuration C_1 and schedule σ applicable to C_1, denoted $run(C_1, \sigma)$, 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 there is no restriction on how often relative to one another processors take steps and increment their clocks, no particular degree of synchronization is necessarily achieved.

A message sent by processor p at event e in infinite schedule σ is *guaranteed* if e is not the last event of σ that involves p. Given configuration C, an infinite run R is *t-admissible* from C, for $0 \leq t \leq n$, if

- $sched(R)$ is applicable to C,
- at most t processors are faulty in R, and
- all guaranteed messages sent to nonfaulty processors in R 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* 0 if there is some processor whose state in C is an element of Y_0; C has decision value 1 if there is some processor whose state in C is an element of Y_1.

2.2 Timing Constraints

We fix a positive constant K to be the number of clock ticks within which a message can be delivered after it is sent and not be considered late. We assume that $K \geq 1$; otherwise messages would always be late, and our model degenerates to that in [FLP]. A message m from p to q is *late* in run R if any processor takes more than K steps between the event when m is sent and the event when m is received (if such an event exists). 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 clock ticks. 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 in terms of the message system delivering enough useful messages, in 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 when p's clock reads K. Asynchronous round r, $r > 1$, begins for p at the end of p's round $r - 1$ and ends either K clock ticks after the end of round $r - 1$, or K clock ticks after p receives the last message sent by a nonfaulty processor q in q's round $r - 1$, whichever happens later.

The reason we require a round to last at least K clock ticks is to prevent a round from collapsing to nothing if no messages are sent in the previous round. This enables processors to make effective use of timeouts. Note that if processors are synchronized, send messages only at the beginning of a round, and all message delays are exactly K, then this definition is the same as the standard synchronous round definition. Thus this definition is not unreasonably strong.

2.3 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 set of n state machines (the n processors without their random number generators and message buffers) and a message pattern, and returns a processor p and a set E of integers 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 was a message sent to p at the i^{th} event), and in no element of the message pattern does p receive this message.

A run is uniquely determined by an adversary A, an initial configuration I, and a collection F of n infinite sequences of random numbers, one sequence for each processor. 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 random number in F for the processor p. Finally, $C_{i+1} = e_i(C_i)$.

If the adversary were not restricted in any way, it could cause all processors 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 collections of n infinite sequences of random numbers F, $run(A, I, F)$ is t-admissible and some nonfaulty processor receives a message in the run.

The expected value of any complexity measure for a fixed randomized protocol is defined as follows. Let T be a function that given a run returns the complexity measure of interest for that run. For fixed 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 to be $\max_{A,I}\{E(T_{A,I})\}$.

2.4 Problem Statement

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

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

The definition of a t-admissible adversary includes the condition that some nonfaulty processor receive a message in order not to penalize a protocol for blocking if no nonfaulty processor ever discovers that it should execute the protocol. However, now there is the possibility of a trivial solution to the problem: if no processor sends any messages, then there is no t-admissible adversary and the t-nonblocking condition is trivially satisfied. To take care of this degenerate case, we add to the definition of t-nonblocking the requirement that in any failure-free run, some processor sends a message.

A protocol is a *transaction commit protocol* if for every t-admissible run R:

- *Agreement Condition:* Every configuration in R has at most one decision value.
- *Abort Validity Condition:* If R is deciding, then whenever the initial value of any processor is 0, then the nonfaulty processors decide 0.
- *Commit Validity Condition:* If R is deciding, then whenever the initial value of all processors is 1 and R is failure-free and on-time, then the nonfaulty processors decide 1.

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

The reason we require a processor to be able to receive at least n messages at one step is to rule out trivial protocols in which nonfaulty processors 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. 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.

For completeness, we give a precise definition of the agreement problem as well. A protocol is an *agreement protocol* if for every t-admissible run R:

- *Agreement Condition:* Every configuration in R has at most one decision value.
- *Validity Condition:* If R is deciding, then whenever the initial value of every processor is 0 (resp. 1), then the nonfaulty processors decide 0 (resp. 1).

3. The Randomized Commit Protocol

Our protocol to solve the transaction commit problem uses a modification of the asynchronous agreement protocol in [Be] as a subroutine. Similar protocols are widely used [Br] [CC] [CMS] [R]. We describe and analyze this agreement protocol first, and then present the complete transaction commit protocol. For the rest of this section, we assume that $n > 2t$.

3.1 Asynchronous Agreement Subroutine

The agreement subroutine is presented as Protocol 1. Each processor p maintains a guess as to the decision in variable x_p, called its *local value*. At the beginning of the protocol, x_p is p's initial value. The protocol is structured

```
Code for processor $p$ in stage $s$, $s \geq 1$:

Input parameters are $x_p$ and $coins$; output parameter
    is agreement value.

broadcast $(1, s, x_p)$

wait to receive $n - t$ messages of the form $(1, s, *)$
    if more than $n/2$ messages are $(1, s, v)$ for some $v$
        then broadcast $(2, s, v)$
        else broadcast $(2, s, \perp)$

wait to receive $n - t$ messages of the form $(2, s, *)$
    if there are no $(2, s, v)$ messages for any $v$
        then $x_p \leftarrow coins[s]$ if $s \leq |coins|$, else $flip(1)$
    if there is a $(2, s, v)$ message for some $v$
        then $x_p \leftarrow v$
    if there are at least $n - t$ messages of the form
            $(2, s, v)$ for some $v$
        then if already decided
            then return($v$)
            else decide $v$
```

Protocol 1: Asynchronous Agreement Subroutine in stages and each stage consists of two sets of message exchanges. At stage s, processor p first sends its local value to everyone and waits to receive $n-t$ of these messages. If more than $n/2$ of them contain some value, v, then p sends v to everyone in the second set of stage s messages; otherwise p sends a special "I don't know" marker. Then p waits for $n-t$ of the second set of stage s messages. If there are at least $n-t$ messages for some value v, then p decides v. If there is at least one message for some v, then p sets its local value to v. Otherwise, p chooses its local value by either referring to a list of coin flips that is initially supplied, or by flipping a local coin.

The "wait" construct used to describe the protocol operates as follows. As a processor receives messages, it posts them on an internal bulletin board. After a wait is encountered in its program, each time a processor takes a step it posts the messages received and then checks if the condition following the wait has been achieved, by looking at all the messages received so far. "Broadcast" means send to all processors, but does not imply atomicity. A processor obtains i random bits by invoking the procedure $flip(i)$.

We establish the correctness and time performance of Protocol 1 for the agreement problem, when called with x_p set equal to p's initial value and with *coins* containing at least n random bits. (Later when we use Protocol 1, we will ensure that *coins* has this property — it is needed for the good time performance, but not correctness.) We call a message of the form $(2, s, v)$, where v is not $\perp$, an *S-message,* because the receipt of such a message causes a processor to set its local variable to v.

Lemma 1: *If every nonfaulty processor's local value is v at the beginning of its stage s, then every nonfaulty processor decides v by the end of its stage s.*

Proof: Since every nonfaulty processor's local value is v, each one sends $(1, s, v)$ at instruction 1 of stage s. Thus at instruction 2, they all receive at least $n-t > n/2$ messages of the form $(1, s, v)$, and they all send $(2, s, v)$. Finally, at instruction 6, they all receive at least $n-t$ messages of the form $(2, s, v)$ and decide v. □

Lemma 2: *During any stage s, there is at most one value sent in S-messages.*

Proof: In order to send an S-message for some value v at stage s, a processor must receive more than $n/2$ messages of the form $(1, s, v)$. Since processors do not send conflicting messages in this fault model, less than $n/2$ messages of the form $(1, s, w)$ for $w \neq v$ can be sent and thus received by a processor. Thus, no processor will send an S-message for w at stage s. □

Lemma 3: *Suppose some nonfaulty processor decides v at stage s. Then every nonfaulty processor also decides v by stage $s+1$.*

Proof: Let r be the earliest stage at which any nonfaulty processor decides, and let p be one of the processors that does so. Without loss of generality, suppose p decides 1.

Now we show that no processor can decide 0 at stage r. Since p decides 1, it receives S-messages for 1. By Lemma 2, there are no S-messages for 0 in stage r, so no processor can decide 0 at stage r.

Now we show that any nonfaulty processor that does not decide at stage r decides 1 at stage $r+1$. Since p decides 1, it receives at least $n-t$ S-messages for 1 at stage r. Thus every other processor receives at least one S-message for 1 at stage r, and sets its local value to be 1. By Lemma 1, they all decide 1 by stage $r+1$.

To complete the proof of the lemma, we consider three cases. If $s = r$, then the lemma follows directly from the preceding argument. If $s = r+1$, then every processor decides v either in stage s or stage $s-1$, and so certainly decides v by stage $s+1$. For any other choice of s, it is not possible for a nonfaulty processor to decide. □

In the remainder of this section we analyze the expected running time of Protocol 1. In particular, we show that it terminates in a small constant expected number of rounds as long as $|coins| \geq n$.

The event we are interested in is that each nonfaulty processor has decided by its stage s, in Protocol 1, denoted $\textsc{decide}(s)$. Another event of interest, $\textsc{same}(s)$, is that all processors that complete stage s set their local values to the same value in stage s. Note that if $\textsc{same}(s)$ occurs, then $\textsc{decide}(s+1)$ occurs.

We define the quantity $random(p, s)$ for processor p and stage s in $run(A, I, F)$. If p completes stage s, then $random(p, s)$ is the random bit returned in the step corresponding to lines 7 through 15 of Protocol 1. Suppose $r < s$ is the latest stage that p completes and p took m steps. Then $random(p, s)$ is the $(s+m)^{th}$ element in the random sequence for p. The goal is merely to obtain unused random bits for the analysis.

At each stage s of run R, let $v(s)$ be the value sent in an S-message, if an S-message is sent, otherwise let $v(s)$ be 0. Define the event $\textsc{match}(s)$, by: if $s \leq n$, then the event is $coins[s] = v(s)$; if $s > n$, then the event is $random(p, s) = v(s)$ for all processors p. $\Pr[\textsc{match}(s)] = 1/2$ if $s \leq n$, and $1/2^n$ if $s > n$. Note that $\textsc{match}(s)$ and $\textsc{match}(s')$ are independent events for $s \neq s'$.

Lemma 4: *If $\textsc{match}(s)$ occurs in R, then $\textsc{same}(s)$ occurs in R.*

Proof: *Case 1:* No S-message is sent in stage s of R, so $v(s) = 0$. Thus each (operating) processor uses a random number (from either *coins* or $flip$) to set its local value. Since $\textsc{match}(s)$ occurs, the processor sets its local value to $v(s)$.

Case 2: An S-message is sent in stage s of R. The value is $v(s)$. Any processor in R that uses the S-message to set its local value, sets its local value to $v(s)$. By the same argument as in Case 1, any processor in R that uses a random number to set its local value, also sets its local value to $v(s)$. □

Lemma 5: *In Protocol 1, for any t-admissible adversary A and initial configuration I,*

$$\lim_{s\to\infty} \Pr[\textsc{decide}(s) \text{ is true in } run(A,I,F)] = 1.$$

Proof: First note that

$$\Pr[\textsc{decide}(s)] \geq \Pr[\textsc{match}(1) \vee \ldots \vee \textsc{match}(s-1)].$$

The reason is that if the event on the right-hand side occurs, then there is an s' between 1 and $s-1$ such that $\textsc{match}(s')$ occurs. By Lemma 4, $\textsc{same}(s')$ occurs, and thus $\textsc{decide}(s'+1)$ occurs.

We next use the fact that the events $\textsc{match}(s)$ are independent.

$$\begin{aligned}
&\Pr[\textsc{match}(1) \vee \ldots \vee \textsc{match}(s-1)]\\
&\quad = \Pr[\neg(\neg\textsc{match}(1) \wedge \ldots \wedge \neg\textsc{match}(s-1))]\\
&\quad = 1 - \Pr[\neg\textsc{match}(1) \wedge \ldots \wedge \neg\textsc{match}(s-1)]\\
&\quad = 1 - \prod_{i=1}^{s-1}(1 - \Pr[\textsc{match}(i)])\\
&\quad \geq 1 - (1 - 1/2^n)^{s-1}
\end{aligned}$$

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

The next lemma shows that each stage takes only a bounded number of asynchronous rounds.

Lemma 6: *In Protocol 1, if each nonfaulty processor is in at most asynchronous round r when it starts stage s, then each is in at most asynchronous round r + 2 when it starts stage s + 1.*

Proof: All $(1, s, *)$ messages sent by nonfaulty processors are at most round r messages by assumption. No nonfaulty processor p can enter round $r+1$ until it has received the last of the round r messages, including all the $(1, s, *)$ messages. Immediately after receiving the last of these (if not before), p sends its $(2, s, *)$ messages, so all $(2, s, *)$ messages sent by nonfaulty processors are at most round $r+1$ messages. No nonfaulty processor p can enter round $r+2$ until it has received the last of the round $r+1$ messages, including all the $(2, s, *)$ messages. Immediately after receiving the last of these (if not before), p sends its $(1, s+1, *)$ messages, so these messages are at most round $r+2$ messages. □

Here is the main theorem showing the correctness of Protocol 1.

Theorem 7: *Protocol 1 is a t-nonblocking consensus protocol.*

Proof: The t-nonblocking property follows from Lemmas 5 and 6. The agreement condition follows from Lemma 3. The validity condition follows from Lemma 1. □

The next lemma is used in the time analysis of our transaction commit protocol.

Lemma 8: *In Protocol 1, all nonfaulty processors decide in a constant expected number of stages.*

Proof: Let $q_s = \Pr[\neg\textsc{match}(s)]$ and let X be the number of stages needed for all processors to decide. Let Y be the number of stages needed for all processors to have the same local value. Since $X \leq Y+1$, EX, the expected value of X, is

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

For $1 \leq s \leq n$, $q_s = 1/2$, and for later stages $q_s = 1 - 1/2^n$.

$$\begin{aligned}
EX &\leq 2 + \left(\sum_{s=1}^{n} q_1 \cdots q_s\right) + \left(q_1 \cdots q_n \cdot \sum_{s=n+1}^{\infty} q_{n+1} \cdots q_s\right)\\
&= 2 + \left(\sum_{s=1}^{n} \frac{1}{2^s}\right) + \left(\frac{1}{2^n} \cdot \sum_{s=n+1}^{\infty} \left(1 - \frac{1}{2^n}\right)^s\right)\\
&< 2 + 1 + \frac{1}{2^n}(2^n - 1)\\
&< 4
\end{aligned}$$

□

3.2 Transaction Commit Protocol

Our transaction commit protocol is presented as Protocol 2. Throughout the protocol each processor keeps a *vote* telling what it currently wants to do with the transaction. Abort corresponds to 0 and commit to 1. The processor with *id* 0 is the *coordinator*, a distinguished processor responsible for beginning the protocol. It flips n coins and sends the result (a random string of n 0's and 1's) around in GO messages to all the processors. Recall that we assume that the adversary cannot see the contents of messages. Once a processor receives a GO message, it relays it to indicate "I am participating in the protocol." If a processor does not receive a GO message from everyone within a short period of time, it changes its vote to abort (if it had previously been commit). Then each processor broadcasts its vote. At this point, any processor that has abort as its vote can actually implement the abort. If a processor receives n commit votes within a short time, it uses 1 as its input x_p to Protocol 1, otherwise it uses 0 as its input. The other input is the GO message, which contains the n coin flips. Then the processor calls Protocol 1. If Protocol 1 returns 1, then the processor commits the transaction, and if 0 is returned, the processor aborts the transaction.

Although our code does not explicitly include it, an important part of the protocol is that GO messages are piggybacked on every message sent, including those of Protocol 1. Thus as soon as a processor (other than the coordinator) receives a message, it has received a GO message.

```
    Code for processor $p$ with initial state $(id, inital)$;
    initially $vote \leftarrow inital$ (1 for commit, 0 for abort):
1.  if $id = 0$ then call $flip(n)$ and broadcast results in
        GO message
2.  else wait for a GO message
3.      broadcast GO
4.  wait for $n$ GO messages or 2K clock ticks
5.  if have not received $n$ GO messages
6.      then $vote \leftarrow 0$
    Choose input to Protocol 1
7.  broadcast $vote$
8.  wait for $n$ $vote$ messages or 2K clock ticks
9.  if received $n$ $vote$ messages for commit
10.     then $x_p \leftarrow 1$
11.     else $x_p \leftarrow 0$
12. call Protocol 1 with $x_p$ and GO message
13. if Protocol 1 returns 1
14.     then decide COMMIT
15.     else decide ABORT
```

Protocol 2: Randomized Transaction Commit Protocol

Theorem 9: *Protocol 2 is a t-nonblocking transaction commit protocol.*

Proof: In order to be precise, we need to take care of the fact that Protocol 2 calls Protocol 1 as a subroutine, and thus not from an initial state, whereas the behavior of Protocol 1 was analyzed for running from an initial configuration. But note that when Protocol 1 is called in Protocol 2, there are no messages in the buffers that can be confused with messages of Protocol 1, every processor has an initial value of 0 or 1, and every processor knows that it is beginning Protocol 1.

Since we are only considering t-admissible adversaries, some nonfaulty processor p does receive a message. Since every message has the GO message piggybacked on it, p now broadcasts GO. Consequently, eventually every nonfaulty processor will receive a GO message, and at most $4K$ clock ticks later it will begin Protocol 1. Protocol 1 is t-nonblocking by Theorem 7.

There are three parts to showing the protocol solves the transaction commit problem. First, there is at most one decision value because Protocol 1 satisfies the agreement condition for the agreement problem. Second, suppose one processor's initial vote is to abort. Then at instruction 7 of Protocol 2 it does not broadcast commit. Thus no processor receives n commit votes during instruction 8 and every processor's input to Protocol 1 is 0. By validity of Protocol 1, every processor decides 0, and at instruction 15 of Protocol 2, every processor aborts.

Finally, suppose every processor initially wants to commit, and the run is failure-free and on-time. We need to show that all processors commit. The coordinator broadcasts GO at time 0 on its clock. By time K on each processor's clock, all processors receive the coordinator's GO and broadcast GO. By time $2K$ on each processor's clock, all processors receive n GO messages. Thus at instruction 7, all processors broadcast 1 as their *vote* messages.

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 *vote* as soon as it receives its n^{th} GO message. Suppose its clock reads T then. Since the run is on-time, every other processor receives its n^{th} GO 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 instruction 10 is executed, setting x_p to 1. By validity of Protocol 1, every processor decides 1, and at instruction 14 of Protocol 2, every processor commits. □

Theorem 10: *In Protocol 2, all nonfaulty processors decide in a constant expected number of asynchronous rounds.*

Proof: An arbitrary nonfaulty processor p receives its first message when it is in at most asynchronous round 2, and begins Protocol 1 at most $4K$ clock ticks after waking up. Since each of p's asynchronous rounds lasts at least K clock ticks (as measured on p's clock), p begins Protocol 1 in at most asynchronous round 6. By Lemma 6, when p begins stage s of Protocol 1, it is in at most asynchronous round $2(s-1)+6$. The expected number of stages of Protocol 1 is 4, by Lemma 8. Now the total is up to 12. Finally, in at most two more asynchronous rounds processors return from Protocol 1 and decide the fate of the transaction. Therefore all nonfaulty processors decide in 14 expected asynchronous rounds. □

Theorem 11: *If more than t processors fail during a run of Protocol 2, no two nonfaulty processors will make conflicting decisions.*

Proof: Suppose more than t processors fail in a run of Protocol 2, and in contradiction that some nonfaulty processor p decides 0 and nonfaulty processor q decides 1.

First we show that p and q cannot return from Protocol 1 at the same stage. If they do, say at stage s, then at stage $s-1$ p receives at least $n-t$ messages of the form $(2, s-1, 0)$ while q receives at least $n-t$ messages of the form $(2, s-1, 1)$. But this is not possible in the fail-stop fault model.

Without loss of generality, assume that p returns at stage s, and q has not yet returned. Since p returns at stage s, p receives at least $n-t$ messages of the form $(2, s-1, 0)$ at stage $s-1$. Pick any nonfaulty processor r. If r receives $n-t$ messages at instruction 6 of stage $s-1$, then r receives at least one message of the form $(2, s-1, 0)$ and sets its local value to 0. (If r does not receive $n-t$ messages at instruction 6, it waits forever.) Thus all messages sent at the beginning of stage s are of the form $(1, s, 0)$. If r receives $n-t$ messages at instruction 2 of stage s, then, since they are all of the form $(1, s, 0)$, r broadcasts $(2, s, 0)$. (If r does not receive $n-t$ messages at instruction 2, it waits forever.) Thus all messages sent at the middle of stage s are of the form $(2, s, 0)$. If q receives $n-t$ messages at instruction 6 of stage s, then, since they are all of the form $(2, s, 0)$, q decides 0. If q does not receive $n-t$ messages at instruction 6, it waits forever. If q receives $n-t$ messages at instruction 2 of stage $s+1$, then it returns 0, otherwise it waits forever.

We have shown that if p returns 0 then q either returns 0 or never returns. □

We make the following remarks in passing. (1) If the run is failure-free and on-time, all the processors decide within at most $8K$ clock ticks, $4K$ for Protocol 2 before calling Protocol 1, and at most $2K$ for each stage of Protocol 1. (2) When the run is on-time (but not necessarily failure-free), the expected number of clock ticks to termination is a constant. (3) By having the coordinator flip more than n coins, the expected value in Lemma 8 can get arbitrarily close to 3 and thus Protocol 2 can terminate in close to 12 expected 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. We first give relevant details of this stronger model.

A processor failure is represented by an explicit *failure step*, denoted $(p, \perp, f)$. 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.)

Processors take steps in round-robin order, p_1 through p_n; a schedule of the form $(p_1, M_1, f_1) \ldots (p_n, M_n, f_n)$ 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 $turn = 1$. In order for an event $e = (p, *, f)$ 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).

For purposes of our lower bound proofs, we assume that the cycle when a message is sent is appended to it. 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 cycle number appended to m. To model the lockstep synchrony of processors, we require that all messages have delay at least 1.

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 proof is similar to that for the coordinated attack problem (see for example [HM]).

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 involving processors in S. Also define $kill(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, f)$ (where $*$ can be M or $\perp$) with $(p, \perp, f)$ whenever p is in S; similarly, define $deafen(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, f)$ (where $*$ can be M or $\perp$) with $(p, \emptyset, f)$ whenever p is in S.

Lemma 12: *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 coin flips. □

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 13: *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 from S' to S that is received 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 in e' p receives no messages. 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 $e = (p, M, f)$ 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 $buff(p, C) \subseteq buff(p, D)$, then m is also 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 12, $state(q, \sigma''(C)) = state(q, \tau''(D))$. 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. □

Theorem 14: There is no t-nonblocking transaction commit protocol if $n \leq 2t$.

Proof: Suppose $n = 2t$ and that there is a t-nonblocking transaction commit protocol with processors p_1 through p_n.

Let $A = \{p_1,\ldots,p_t\}$ and $B = \{p_{t+1},\ldots,p_n\}$. The first t events of a cycle form an *A-semicycle* (each processor in A takes a step); the last t events of a cycle form a *B-semicycle* (each processor in B takes a step). Note that an infinite schedule applicable to an initial configuration consists of alternating A- and B-semicycles. Define a *phase* to be a schedule consisting of one or more semicycles in which all intergroup messages received (if any) flow in the same direction (either from A to B, or from B to A).

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 $\alpha(I_{11})$ has decision value 1.

Let $\alpha = \pi_1 \ldots \pi_y$ where each π_i is a phase and, for all $1 \le i \le y-1$, the intergroup messages received in π_i flow in the opposite direction from those received in π_{i+1}. (It does not matter if such a partition of α is not unique.)

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

Proof of Claim: We show the claim by descending induction on i.

Basis: $i = y+1$. Letting $\alpha_{y+1} = \alpha$ proves the claim.

Induction: $i < y+1$. We assume the claim is true for $i+1$ and show it for i. Without loss of generality, assume intergroup messages received in π_i flow from A to B. (See Figure 1.) Define β_1 to be $deafen(B, \pi_i\gamma_{i+1})$. By Lemma 13, β_1 is applicable to C_{i-1}. Since $\beta_1|A = \pi_i\gamma_{i+1}|A$, Lemma 12 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 by F. No intergroup message is received in β_1 because processors in B receive no messages and processors in A receive no intergroup messages in $\pi_i\gamma_{i+1}$.

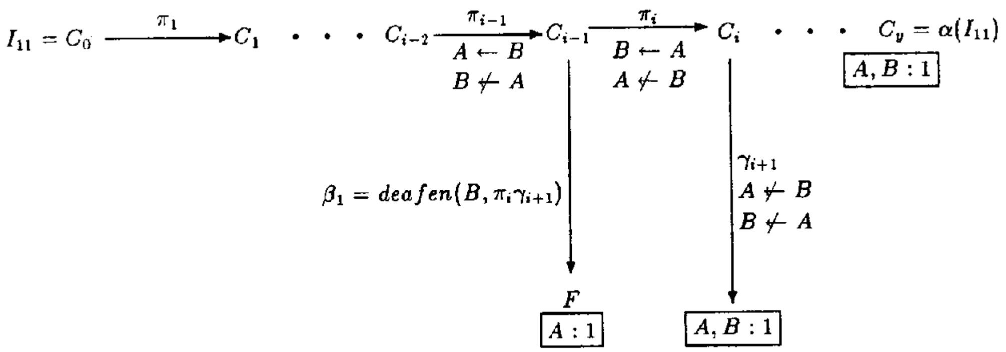

Figure 1: $run(\alpha_i, I_{11})$

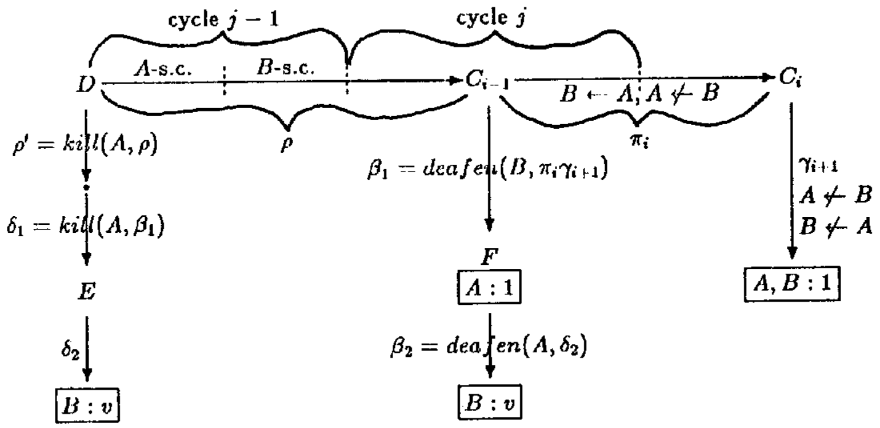

Figure 2: Constructing α_i from α_{i+1}

Suppose the first semicycle of π_i is part of the j^{th} cycle of 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 ρ be the substring of α_i between D and C_{i-1}; ρ consists of an A-semicycle followed by a B-semicycle, and possibly another A-semicycle. Let $\rho' = kill(A, \rho)$. By choice of α and ρ, any message received in ρ' by a processor p in B from a processor in A was sent prior to cycle $j-1$ and is in $buff(p, D)$. By Lemma 13, ρ' is applicable to D. Since $\rho|B = \rho'|B$, Lemma 12 implies that $state(p, \rho'(D)) = state(p, C_{i-1})$ for all p in B.

Consider the schedule $\delta_1 = kill(A, \beta_1)$. (See Figure 2.) Since the processors in A fail and the processors in B receive no messages, δ_1 is obviously applicable to $\rho'(D)$. Let $E = \delta_1(\rho'(D))$. Since $\delta_1|B = \beta_1|B$ and $state(p, \rho'(D)) = state(p, C_{i-1})$ for all p in B, Lemma 12 implies that $state(p, E) = state(p, F)$ for all p in B.

By the t-nonblocking property, there must exist a finite deciding run from E with schedule δ_2. Suppose the decision value is v. By choice of α, all messages sent before cycle $j-1$ are received by the end of cycle j in ρ. Since $\rho'|B = \rho|B$, every processor in B receives in ρ' all messages sent to it before cycle $j-1$. Thus in δ_2, processors in B receive messages sent at cycle $j-1$ or later. Since all processors in A have been dead since cycle $j-1$, B receives no intergroup messages in δ_2.

Let $\beta_2 = deafen(A, \delta_2)$. Pick p in B. From above, $state(p, E) = state(p, \beta_1(C_{i-1}))$. Let m be any message in $buff(p, E)$; m could only have been sent by a processor q in B in cycle $j-1$ or later. Lemma 12 implies that q has the same state in corresponding configurations in $run(\rho'\delta_1, D)$ 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 13 to show that β_2 is applicable to F.

Since $\beta_2|B = \delta_2|B$ and $state(p, F) = state(p, E)$ for all p in B, Lemma 12 implies that each processor p in B is in the same state in $\beta_2(F)$ as in $\delta_2(E)$. So B decides v in $\beta_2(F)$; by the agreement condition, $v = 1$, because processors in A have already decided 1 by F. No intergroup messages are received in β_2 because none are in δ_2.

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 13, σ is applicable to I_{11}. Since $\sigma|B = \alpha_1|B$, Lemma 12 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 13, 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 12 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 by the abort validity condition as well as the t-nonblocking property, $\sigma(I_{01})$ must have decision value 0, which is a contradiction. □

5. Lower Bound on Time

In this section we prove that no protocol can terminate in a constant expected number of clock ticks. This result provides additional justification for our definition of asynchronous rounds, and says that in some sense our protocol has "optimal" time performance.

For the result of this section to hold, we must make a technical restriction on the class of possible protocols. We assume that for any protocol P, there is a function f such that for any processor p and any step s, processor p uses at most $f(s)$ random bits at step s in any run of protocol P. We need the following definitions in addition to the definitions and Lemmas 12 and 13 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* is a set of n sequences of random numbers such that either each sequence is infinite or each sequence has the same number of elements, and there is a one-to-one correspondence between processors and sequences. The *length* of F is the length of one sequence.

A run is *F-compatible*, for seed F, if for all processors p and all i not exceeding the length of F, when p's clock reads i, the random number that p receives is the i^{th} element of p's sequence in F. Given configuration C, a schedule σ is *F-compatible relative to C* if $run(C, \sigma)$ is F-compatible.

For the remainder of this section, we fix an arbitrary 1-nonblocking transaction commit protocol. We are only concerned with configurations reachable from some initial configuration by a 1-admissible run.

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.

Lemma 15: *Choose some integer x and some finite seed F, and let I_1 be the initial configuration in which all processors have initial value 1. If $run(I_1, \tau)$ is a finite failure-free on-time deciding run that is F-compatible, then there exists a configuration in $run(I_1, \tau)$ that is $(x, F, \{0,1\})$-valent.*

Proof: Pick such a run $run(I_1, \tau)$. 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 0-valent.

By Lemma 13, σ is also applicable to I_1. By Lemma 12, all processors except q have the same state in $\sigma(I_1)$ as in

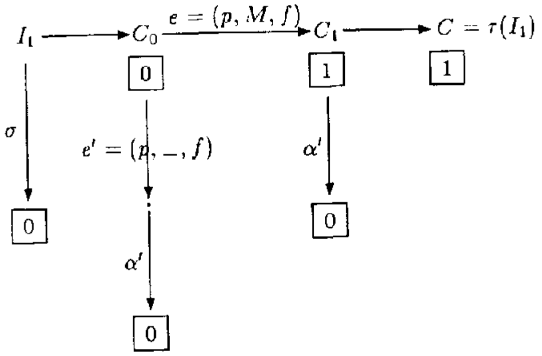

Figure 3: Demonstrating the existence of an $(x, F, \{0,1\})$-valent configuration

$\sigma(I_{01})$, and decide 0 in $\sigma(I_1)$. Thus I_1 is either $(x, F, \{0\})$-valent or $(x, F, \{0,1\})$-valent.

The valencies of I_1 and C imply that there must be an event $e = (p, M, f)$ 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.)

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 $\alpha = (p, \perp, f)\alpha'$. Since α' is applicable, F-compatible and x-slow relative to C_1, and C_1 is $(x, F, \{1\})$-valent, all the nonfaulty processors decide 1 in $\alpha'(C_1)$. But all the processors except p have the same state in $\alpha'(C_1)$ as they do in $\alpha(C_0)$ (by Lemma 12), where they decide 0. This is a contradiction. □

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

Lemma 16: *Choose a finite failure-free run R' that decides 1 and has all message delays equal to 1. Let F' be the finite seed of R' and let y be the length of R'. For any $x > 0$, choose a seed F of length $y + x$ that extends F'. Let C be an $(x, F, \{0,1\})$-valent configuration in R'. Then there is a finite F-compatible run R containing C such that $T(R'') > x$ for any infinite run R'' which is an extension of R.*

Proof: First note that C exists by Lemma 15. Let $C = \alpha(I)$. Consider the failure-free x-cycle schedule σ that is applicable and F-compatible relative to C in which no processor receives a message. We show that $\sigma(C)$ is $(x, F, \{0,1\})$-valent. The lemma follows by letting $R = run(I, \alpha\sigma)$.

Without loss of generality, assume $\sigma(C)$ is $(x, F, \{0\})$-valent. Then there is a configuration D in $run(\sigma, C)$ and some event $e = (p, M, f)$ in σ such that D is $(x, F, \{0,1\})$-valent and $e(D)$ is $(x, F, \{0\})$-valent. The only other event applicable to D that can be part of an x-slow F-compatible run is $(p, \perp, f) = e'$, 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.) 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 to $e(D)$, and all processors except p have the same state in $\tau(e(D))$ as in $\tau(e'(D))$, so they decide 1, contradicting the valency of $e(D)$. □

Theorem 17: *For any constant B, there is an adversary A and an initial configuration I such that $E(T_{A,I}) \geq B$.*

Proof: Let A' be the adversary that kills no processors and sets all message delays to 1. Let I_1 be the initial configuration in which all initial values are 1. Let $\mathcal{R}$ be the set of all $2B$-cycle failure-free runs from I_1 such that the message delay for all messages is 1. There is a finite number of such runs.

Case 1: At most half the runs in $\mathcal{R}$ are deciding. Let $A = A'$ and $I = I_1$. Then $E(T_{A,I}) \geq 2B/2 = B$.

Case 2: More than half the runs in $\mathcal{R}$ are deciding. Let $\mathcal{C}$ be the set of all configurations present in some run in $\mathcal{R}$, and let $m = |\mathcal{C}|$. Keep a count for each C in $\mathcal{C}$, initially 0. Let $\mathcal{F}$ be the collection of all seeds with length $2mB$. $\mathcal{F}$ is finite by the technical assumption made that at each step a processor uses only a finite number of random bits.

For each F in $\mathcal{F}$ do the following. Let R be the F-compatible run in $\mathcal{R}$. If R is not deciding, do nothing. If R is deciding, then by Lemma 15, there is a $(2mB, F, \{0,1\})$-valent configuration C reachable from an initial configura-

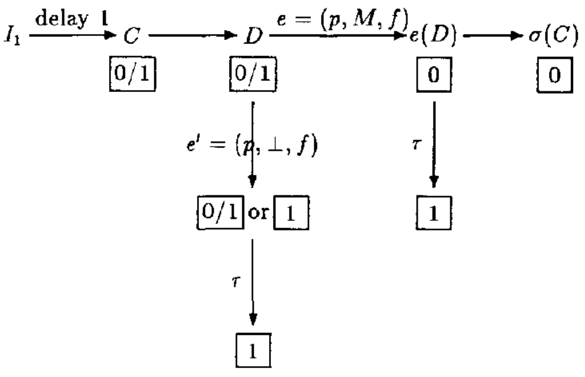

Figure 4: Demonstrating that $\sigma(C)$ is $(x, F, \{0,1\})$-valent

tion by some failure-free run R'' with delays 1. Thus C is in $\mathcal{C}$. Let l be the length of R''. Increment C's count by 1. By Lemma 16, there is a finite F-compatible run R' containing C such that $T(R'') > 2mB$, for any infinite run R'' which is an extension of R'. Let A_C be the adversary of R'. That is, A_C is the adversary which for the first l events delivers messages after delay 1 and which subsequently delivers messages after delay $2mB$.

Since $|\mathcal{C}| = m$, there is a C in $\mathcal{C}$ with count at least $\frac{1}{m} \cdot \frac{1}{2} \cdot |\mathcal{F}|$, because of the pigeonhole principle and the fact that at least half the elements of $\mathcal{F}$ cause a count to be incremented.

Let $I = I_1$ and $A = A_C$. Then

$$E(T_{A,I}) \geq \frac{1}{m} \cdot \frac{1}{2} \cdot 2mB = B$$

because the fraction of all runs from I with adversary A that contain C is at least $1/2m$ and the value of T for each of those runs is at least $2mB$. □

Acknowledgment

We would like to thank Barbara Liskov, Nancy Lynch, and Bill Weihl for suggesting this problem to us.

References

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

[Br] G. Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Algorithm," *Proc. 17th 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.

[CMS] B. Chor, M. Merritt, and D. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models," *Proc. 4th 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," *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Science,* pp. 393–402, 1983.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Proc. 3rd 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.

[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.)