Distributed Computing (1990) 4:87–103

DISTRIBUTED COMPUTING

Transaction commit in a realistic timing model*

Brian A. Coan[1] **and Jennifer Lundelius Welch**[2]

[1] Bellcore, Morristown, NJ 07960, USA
[2] University of North Carolina, Chapel Hill, NC 27599, USA

Received June 12, 1989 / Accepted March 3, 1990

Brian A. Coan received the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, New Jersey, in 1977; the M.S. degree in computer engineering from Stanford University, Stanford, California, in 1979; and the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, Massachusetts, in 1987. He has worked for Amdahl Corporation and AT & T Bell Laboratories. Currently he is a member of the technical staff at Bellcore. His main research interest is fault tolerance in distributed systems.

Jennifer Lundelius Welch received her B.A. in 1979 from the University of Texas at Austin, and her S.M. and Ph.D. from the Massachusetts Institute of Technology in 1984 and 1988 respectively. She was a member of technical staff at GTE Laboratories Incorporated in Waltham, Massachusetts, from 1988 to 1989. She is currently an assistant professor at the University of North Carolina in Chapel Hill. Her research interests include algorithms and lower bounds for distributed computing.

* The authors were with the MIT Laboratory for Computer Science when the bulk of this work was done. This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract N00014-83-K-0125, the National Science Foundation under Grant DCR-83-02391, the Office of Army Research under Contract DAAG29-84-K-0058, and the Office of Naval Research under Contract N00014-85-K-0168. A preliminary version of this paper appears in the *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* [2]

Offprint requests to: J. Lundelius Welch

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 described. Developed for this model is a new nonblocking randomized transaction commit protocol, which incorporates an agreement protocol of Ben-Or. The new protocol works as long as fewer than half the processors fail. A matching lower bound is proved, showing that the number of processor faults tolerated is optimal. If half or more of the processors fail, the protocol degrades gracefully: it 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. In contrast it is shown that no protocol in this model can guarantee that a processor terminates in a bounded expected number of its own steps, even if processors are synchronous.

Key words: Distributed databases – Fault tolerance – Lower bounds – Randomized protocols – Time bounds – Transaction commit

1 Introduction

In a distributed database system a transaction may be processed concurrently by 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 should be installed in the database at all processors (the transaction is *committed*), or the results should be installed at no processor (the transaction is *aborted*). The objective of a *transaction commit protocol* is to ensure that consistent

action is taken and that as many transactions as practicable are committed. The protocol is subject to the constraint that each processor must be able to abort a transaction unilaterally (i.e., if any processor wishes to abort, the decision must be abort).

The definition of the transaction commit problem allows some leeway in protocol design regarding which circumstances require the decision to be commit. To avoid useless protocols that abort all transactions, it is usual to impose the additional requirement that a protocol must commit a transaction in any failure-free execution in which all processors vote commit. In this paper, we demonstrate the benefit of relaxing that requirement slightly.

Another variation among protocols is that some may fail to terminate in certain situations. If failures cause some nonfaulty processor to remain undecided about the fate of a transaction (at least as long as the failures persist), that 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. In the presence of processor failures, a blocking protocol can delay transaction processing for a long time, although it does allow correct action to be taken after the failure is repaired. The impact of processor failures is somewhat less with a nonblocking protocol.

Elegant nonblocking transaction commit protocols have been developed for completely synchronous systems by Skeen [12] and Dwork and Skeen [6]. 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 fail, producing 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 describe a new timing model that is intermediate between the synchronous and asynchronous models previously studied. In this model, we develop a new nonblocking transaction commit protocol.

We model real systems in which messages are usually delivered within some known time bound, but sometimes come late. Our approach 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 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. (Dwork, Lynch, and Stockmeyer [5] make a similar division between properties that must always hold and properties that only need hold when the system is well-behaved. In most other respects our model differs from theirs.)

In contrast, Dwork and Skeen [7] study the transaction commit problem in a completely asynchronous model in which processor failures are detectable, i.e., always announced in finite time. In this model, they are able to take advantage of the failure notification to design efficient nonblocking transaction commit protocols.

We assume that the faulty processors fail by crashing (i.e., stopping without warning). Our protocol works as long as more than half the processors are nonfaulty. The number of faults tolerated by our protocol is optimal, as shown by the matching lower bound that we prove. (The synchronous protocols of Skeen [12] and Dwork and Skeen [6] tolerate any number of processor faults.) An important property of our protocol is that it degrades gracefully: if the bound on the number of faulty processors is exceeded, the protocol simply fails to terminate instead of producing a wrong answer.

Our protocol uses a modified version of a solution to the agreement problem. The agreement problem and the transaction commit problem, although superficially similar, are different problems. 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 an acceptable decision for the agreement problem, whereas in the transaction commit problem, the decision 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 [12] has an excellent discussion of recovery protocols. We do not discuss these protocols further in this paper. Although the definition of the agreement problem places no constraints on the decisions reached by faulty processors, some agreement protocols have the property that even decisions reached

[1] The outline of this impossibility result is the following. Suppose there is a protocol that works in an asynchronous system and guarantees that (1) nonfaulty processors eventually decide (with probability 1); (2) if the processors all begin with commit and there are no failures, then they all decide commit; and (3) 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. 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. In this run, the decision must be commit. All processors except p have the same view in the two runs but must reach different decisions, contradicting the assumed existence of the protocol

by faulty processors are correct. Our transaction commit protocol incorporates one such agreement protocol.

In our protocol, processors exchange some messages and then execute a modification of the asynchronous agreement protocol of Ben-Or [1] 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 [2]. 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 problem (see for example Rabin [11] or Chor, Merritt, and Shmoys [3]). Our technique does not solve this problem because of the following difference between the agreement and transaction commit problems. In our protocol, if the identical random bits are not distributed in a timely fashion, processors can unilaterally decide 0 (abort) and still satisfy the conditions of the transaction commit problem. Such an action is not an option for processors in an agreement protocol, because it could violate the condition that all processors decide 1 if they all start with 1.

Randomization is needed in our protocol because the well-known result of Fischer, Lynch, and Paterson [8] implies that no deterministic protocol is possible. In order to analyze a randomized protocol, we must define the adversary against which the protocol will work. Our notion of the adversary is inspired by Chor, Merritt, and Shmoys [3]. 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 information is deducible from the pattern of communication. We will be careful to design our protocol so that it is not deducible.

We prove that in our model no transaction commit protocol can guarantee that each processor terminates in a bounded expected number of its own steps, even if processors are synchronous and only a single fault is to be tolerated. 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 protocol terminates in a small constant expected number of asynchronous rounds.

[2] We have not solved the global coin toss problem, however, because our protocol does not guarantee that the identical random bits are successfully distributed. Fortunately, the unique nature of the transaction commit problem allows us to design a protocol in which a processor only needs to consult these bits in those executions in which they have been successfully distributed

Following an exposition of our formal model in Sect. 2, we present and analyze our randomized transaction commit protocol in Sect. 3. Section 4 contains the lower bound proof showing that our protocol tolerates the maximal number of faulty processors. In Sect. 5 we show that no transaction commit protocol can guarantee that each processor terminates in a bounded expected number of its own steps, even if processors are synchronous and only a single fault is to be tolerated. Section 6 contains a summary.

2 Model

There are n processors that are to decide the fate of a particular transaction. (Our protocol assumes that $n \geq 1$; our lower bounds assume that $n \geq 2$, and are not true if $n = 1$.) 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 bounds are shown in the stronger model in which processors run in lockstep synchrony and possess atomic broadcast. In this section we present the weaker model. In Sects. 4 and 5 we indicate the necessary changes for the stronger model. Our model is similar to those of Fischer, Lynch, and Paterson [8] and Dolev, Dwork, and Stockmeyer [4].

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 an ordered pair (raw message, integer); 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 (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. Certain processor states are initial states, designated (*id*, *initval*), where *id* is a nonnegative integer and *initval* is either 0 or 1. The *id* element of an initial state is the processor's name, or identification number. The *initval* element is the processor's initial value. There is an integer in each processor's state, called its *clock*, which is 0 in all initial states. The state machine's transition function is applied to a state, an n-bit string, and a set of raw messages to produce another state and a set of raw messages containing

at most one raw message per recipient. The transition function always increments *clock* by 1.

Described informally, a processor at each step computes a new state and a set of raw messages to send, based on its current state, the set of raw messages just received, and an n-bit string from its random number generator. The processor keeps track of how many steps it has taken with the *clock* variable.

A *protocol* is a set of n processors, with identification numbers 0 through $n-1$. A particular protocol is implicit in all the definitions in the remainder of Sect. 2.

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. Such an event is a *step* of p.

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 the triple consisting of 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 recipient's 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 $C_{i+1} = e_i(C_i)$, $1 \le i \le k$. 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 $i \ge 1$, $C_{i+1} = e_i(C_i)$. 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. Note that processors cannot be designated as faulty or nonfaulty in a finite run or schedule.

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 *admissible* if the first configuration is an initial configuration 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. An admissible run is *t-admissible* for $0 \le t \le n$, if at most t processors are faulty in the run.

Each processor's state set contains two disjoint subsets of *decision states*, Y_0 and Y_1, such that the transition function applied to a state in Y_v produces a state in Y_v (i.e., once a processor enters a state in Y_0 or Y_1 in a run, it stays in that set forever). A processor *decides* v in a run when it is in a state in Y_v. A configuration C has *decision value* v if there is some processor whose state in C is an element of Y_v. An infinite run is *deciding* if every nonfaulty processor decides.

2.2 Timing constraints

We fix a positive constant $K \ge 1$, which is used to define late messages. A message m from p to q is *late* in (finite or infinite) run $R = C_1\, e_1\, C_2\, e_2 \ldots$ if event e_s adds m to q's message buffer, event e_r removes m from q's message buffer, and there is some processor that 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. Note that a message that is never delivered is not considered late.

Ideally we would like a processor to decide in a constant expected number of its own steps. Unfortunately, as we prove in Sect. 5, this is impossible, even if processors run in lockstep synchrony and only a single fault is to be tolerated. 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 for processor p consists of p's first K steps. Asynchronous round r, $r > 1$, for p begins with the first step that p takes after the end of p's round $r-1$. It ends with the first step in which p has satisfied all of the following three conditions: p has taken at least K steps in round r, p has received every guaranteed message that was sent by a processor q in q's round $r-1$, and in the remainder of the infinite run there are no steps in which p receives a message that was sent by a processor q in q's round $r-1$. (Note that the last two conditions make sure that no round lasts infinitely long due to p's waiting for a non-guaranteed message 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 ommiscient observers as we analyze protocols. We require a round to last at least K steps 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 of Dwork and Skeen [6].

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 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.

Since these three conditions must hold for any admissible run, regardless of how many processors are faulty, our definition of transaction commit protocol incorporates the graceful degradation property: processors may block but will never produce the "wrong" answer.

The definitions in Subsect. 2.1 allow each processor to receive an unbounded number of messages at each step. This assumption is not essential to our work, but to exclude uninteresting protocols, we must require that each processor 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[3]. 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 which 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 $e_i = (p_i, M_i, b_i)$ 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.

[3] For an example of swamping, consider the following. 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 failure-free run is on-time. Let R be an infinite failure-free 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

An *adversary* is a function that takes a message pattern $(p_1, E_1, P_1) \ldots (p_k, E_k, P_k)$ and returns a processor p, that will take step $k+1$, and a set of at most n messages sent during the first k events whose receipt is delayed until the $k+1^{st}$ event. This set of messages is represented by a set E of integers, $1 \leq |E| \leq n$, such that for all $i \in E$, $p \in P_i$.

Let $\mathcal{F}$ be the collection of all n-tuples of infinite sequences of n-bit strings. Each element of $\mathcal{F}$ is an n-tuple $(x_0, \ldots, x_{n-1})$, where for all p, x_p models the sequence of random strings that could be returned by processor p's random number generator in p's steps in some 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. Event e_i is applicable to C_i by the definition of an adversary. We define C_{i+1} to be $e_i(C_i)$. 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, for $0 \leq t \leq n$, 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 is defined as follows. Let T be a random variable that, for a given run, is the value of the complexity measure of interest for that run. For a fixed admissible adversary A and initial configuration I, let the expected value of T, taken over all F in $\mathcal{F}$, be denoted $E(T_{A,I})$. Define the expected value for the protocol for a given value of t, $E(T, t)$, to be $\max_{A,I}\{E(T_{A,I})\}$, where A is any t-admissible adversary and I is any initial configuration.

2.5 Liveness condition

Given admissible run R and integer r, let 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,

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

The t-nonblocking property means that the probability of all the nonfaulty processors having decided goes to 1 as the number of rounds increases without bound.

3 The randomized commit protocol

For all of this section we assume a fixed $t \geq 0$ with $n > 2t$. Subsection 3.1 contains the code for our t-nonblocking transaction commit protocol, preceded by an informal description. In Subsect. 3.2 we prove that our protocol is a transaction commit protocol, i.e., it satisfies the safety conditions. In Subsect. 3.3 we prove an eventual termination property which is used in Subsect. 3.4 to show that our protocol is t-nonblocking. Subsection 3.5 contains the time analysis, in which we show that our protocol decides in a constant expected number of asynchronous rounds.

3.1 The protocol

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

Our protocol consists of a few preliminary message exchanges followed by a modification of the agreement protocol of Ben-Or [1]. The two purposes of the preliminary message exchanges are to resolve the differences between the input/output relations of the transaction commit and agreement problems (e.g., to ensure that if any vote is initially 0, then all the inputs to the agreement "subroutine" are 0) and for the coordinator to distribute n random bits to all the processors.

The original protocol of Ben-Or [1] proceeds in stages, with each processor using one random bit at each stage. The protocol is sure to terminate once a stage is reached in which each processor's random bit is equal to a particular value (chosen for that stage by the adversary). Obviously, if each processor's random bit in a stage is independent of every other processor's random bit for that stage, the expected number of stages until termination is exponential in the number of processors. In our protocol, the coordinator distributes n random bits to be used in the first n stages, one bit per stage. Thus all the processors will share the same random bit in each of the first n stages. The probability that none of the n common bits has the required value for its stage is exponentially small, causing the expected number of stages until termination to be constant.

We now describe the two parts of our protocol in more detail.

Throughout the protocol each processor keeps a *vote* indicating 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. (Throughout this paper we use "broadcast" to mean send to all processors.) If a participant receives no message at its first step (which only happens if the participant unilaterally initiates the protocol), 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 either receives a message at its first step or receives a timely reply to its request message, it extracts the n bits and broadcasts them in a coins message, to indicate that it is participating in the protocol. If all processors are nonfaulty and the run is on-time, then each processor receives a coins message from everyone within $2K$ steps after broadcasting one. If a processor does not receive these messages, it sets its vote to 0 and decides 0. In either event, each processor then broadcasts its vote. If a processor does not receive n votes for 1 within an additional $2K$ of its steps, it sets its vote to 0, but remains undecided.

The rest of the protocol proceeds in stages (as in Ben-Or [1]), 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 at least $n-t$ stage $(s, 1)$ messages with the same value $v \in \{0, 1\}$, then p broadcasts v in a stage $(s, 2)$ message; otherwise p broadcasts "?" in a stage $(s, 2)$ message. The purpose of the first part of stage s is to ensure that it is never the case that some processor broadcasts 0 in a stage $(s, 2)$ message and another processor broadcasts 1 in a stage $(s, 2)$ message. In the second part of stage s, processor 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 at least $n-t$ stage $(s, 2)$ messages for value $v \in \{0, 1\}$, then p decides v.

Processor p uses the following constants and variables. Constants, in addition to p itself, are n, t, and K as defined above. 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 ∞; initially ∞.
- $coins_p$: n-bit string or *nil*; initially *nil*.
- $vote_p$: boolean; initially p's initial value.
- $decide_p$: boolean or *nil*; initially *nil*.
- $received_p$: set of raw messages; initially empty.

The text of each raw message consists of either a possible value for a $stage_p$ variable, or a triple containing a possible value for a $stage_p$ variable, an element of $\{0, 1, ?\}$, 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 state of p returned by the transition function is obtained from q in accordance with the following pseudocode. The set of raw messages returned by the transition function is that indicated by the send statements executed in the pseudocode. The statement "**if** expression **then** body **elseif** expression **then** body ... **elseif** expression **then** body **endif**" is a multiway branch.

Protocol 1:

/* increment clock and save received raw messages */

$clock_p := clock_p + 1$
$received_p := received_p \cup M$

```
/* coordinator initiates protocol
      by distributing n random bits */
  if stage_p="asleep" and p is the coordinator then
    coins_p:=b
    stage_p:="coins"
    timer_p:=clock_p+2K
    send (stage_p, "?", coins_p) to all processors

/* non-coordinator wakes up and requests
      that coordinator initiate */
  elseif stage_p="asleep" and p is not the coordinator
      and M=∅ then
    stage_p:="request"
    timer_p:=clock_p+2K
    send (stage_p) to the coordinator

/* non-coordinator receives coins before timeout */
  elseif (stage_p="asleep" and p is not the coordinator
      and M≠∅) or (stage_p="request" and clock_p
      ≤timer_p and M≠∅) then
    coins_p:=n-bit string from any raw message in M
    stage_p:="coins"
    timer_p:=clock_p+2K
    send (stage_p, "?", coins_p) to all processors

/* non-coordinator times out while waiting
      to receive coins */
  elseif stage_p="request" and clock_p
      =timer_p and M=∅ then
    vote_p:=0
    decide_p:=0

/* distributing votes */
  elseif stage_p="coins" and (clock_p=timer_p or n coins
      messages are in received_p) then
    stage_p:="vote"
    timer_p:=clock_p+2K
    if fewer than n coins messages are in received_p then
      vote_p:=0
      decide_p:=0
    endif
    send (stage_p, vote_p, coins_p) to all processors

/* completing stage 0 */
  elseif stage_p="vote" and (clock_p=timer_p or n vote
      messages are in received_p) then
    stage_p:=(1, 1)
    if fewer than n vote messages
      for 1 are in received_p then
      vote_p:=0
    endif
    send (stage_p, vote_p, coins_p) to all processors

/* finishing first part of stage s */
  elseif stage_p=(s, 1) and at least n-t stage (s, 1)
      message are in received_p then
    stage_p:=(s, 2)
    if there is v∈{0, 1} such that at least n-t stage
      (s, 1) messages in received_p have value v then
      send (stage_p, v, coins_p) to all processors
    else
      send (stage_p, "?" coins_p) to all processors
    endif

/* finishing second part of stage s */
  elseif stage_p=(s, 2) and at least n-t stage (s, 2)
      messages are in received_p then
    stage_p:=(s+1, 1)
    if there is v∈{0, 1} such that a stage (s, 2) message
      in received_p has value v then
      vote_p:=v   /* in Lemma 4 we show that v is
                     unique */
      if at least n-t stage (s, 2) messages
      in received_p have value v then
        decide_p:=v
      endif
    else
      if s≤n then vote_p:=coins_p [s]
        else vote_p:=the first bit of b endif
    endif
    send (stage_p, vote_p, coins_p) to all processors
  endif
```

3.2 *Safety conditions*

This subsection culminates in Theorem 8, which asserts that Protocol 1 is a transaction commit 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.

Lemma 1. *In any admissible run, if for some* p $vote_p = 0$ *at any time before* p *broadcasts its vote message, then every stage* (1, 1) *message has value* 0.

Proof. Since $vote_p$ is never set to 1 before stage 1, no processor ever receives a vote message with value 1 from p. Thus no processor broadcasts a stage (1, 1) message with value 1. □

Lemma 2. *In any admissible run, if for all* p $vote_p = 1$ *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 thus p receives some message containing the coins (not necessarily from the coordinator) at time $T_1 \leq 2K$ on p's clock. Then p broadcasts a coins message at time T_1; by time T_1+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 T_1+2K 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_2 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_2+K. Thus p receives all n vote messages by the time its clock reads T_2+2K. Then p broadcasts its stage (1,1) message with value 1. □

Lemma 3. *In any admissible run, for all $s \geq 1$, if every stage $(s, 1)$ message has value $v \in \{0, 1\}$ then*
(1) *every stage $(s, 2)$ message has value v;*
(2) *every stage $(s+1, 1)$ message has value v; and*
(3) *any processor p that completes stage s sets $decide_p$ to v at the end of stage s.*

Proof. Part 1 is obvious from the code. Parts 2 and 3 follow from part 1 and the code. □

For any $s \geq 1$, we call a stage $(s, 2)$ message with value $v \in \{0, 1\}$ a *stage s 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 admissible run, for any fixed $s \geq 1$, every stage s S-message has the same value.*

Proof. Fix $s \geq 1$. In order to send a stage s S-message with value v, a processor must receive at least $n-t$ stage $(s, 1)$ messages with value v. Since no processor broadcasts conflicting messages, at most t processors can broadcast a stage $(s, 1)$ message with value $w \neq v$. Thus, no processor receives more than t stage $(s, 1)$ messages with value w. Since $n > 2t$, t is less than $n-t$, and no processor sends a stage s S-message with value w. □

Lemma 5. *In any admissible run, for all $s \geq 1$ and all processors p, if p sets $decide_p$ to v in stage s, then all stage $(s+1, 1)$ messages have value v.*

Proof. Suppose p sets $decide_p$ to v in stage s. Then p receives at least $n-t$ stage s S-messages for v. Let S_p be the set of processors that send stage s S-messages for v. Let q be any processor that completes stage s. Then q receives at least $n-t$ stage $(s, 2)$ messages. Since $n > 2t$, at least one of these $n-t$ messages received by q is from a processor in S_p. Since no processor broadcasts conflicting messages, q receives at least one stage s S-message for v. By Lemma 4, q receives no stage s S-message for any $w \neq v$. Therefore q's stage $(s+1, 1)$ message has value v. □

Lemma 6. *In any admissible run, for all processors p and q (not necessarily distinct) if p sets $decide_p$ to v at some point in the run, and q sets $decide_q$ to w at another point in the run, then $v = w$.*

Proof. Without loss of generality, assume that the designated point for p does not come after the designated point for q. There are three cases.

Case 1. Processor p sets $decide_p$ to v before completing stage 0, and q sets $decide_q$ to w before completing stage 0. By the code, $v = 0 = w$.

Case 2. Processor p sets $decide_p$ to v before completing stage 0, and q sets $decide_q$ to w at the end of stage s, for some $s \geq 1$. By the code, $v = 0$. By Lemma 1, every stage $(1, 1)$ message has value 0. By part 2 of Lemma 3 and induction, every stage $(s, 1)$ message has value v. By part 3 of Lemma 3, $w = v$.

Case 3. Processor p sets $decide_p$ to v at the end of stage r, for some $r \geq 1$, and q sets $decide_q$ to w at the end of stage s, for some $s \geq 1$. By our choice of p and q, $r \leq s$. There are two subcases.

Case 3a. If $r = s$, then p receives at least $n-t$ stage r S-messages for v and q receives at least $n-t$ stage r S-messages for w. By Lemma 4, $v = w$.

Case 3b. Suppose $r < s$. By Lemma 5, all stage $(r+1, 1)$ messages have value v. By part 2 of Lemma 3 and induction, all stage $(s, 1)$ messages have value v. By part 3 of Lemma 3, $w = v$. □

Lemma 7. *In any admissible run, $decide_p$ changes value at most once, for every processor p.*

Proof. By Lemma 6. □

Let p's decision states Y_0 and Y_1 be states with $decide_p = 0$ and $decide_p = 1$ respectively; Lemma 7 shows that once p enters a state in Y_v, $v \in \{0, 1\}$, it stays in that set forever. Thus we can say p *decides* v when p sets $decide_p$ to v for the first time in a run.

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

Proof. By Lemma 7 and inspection, Protocol 1 is actually a protocol, according to our definition. It remains to show that it is a transaction commit protocol.

Let R be a t-admissible run. The agreement condition is satisfied by Lemma 6.

Next we show the abort validity condition. Suppose some processor begins with initial value 0. By the code, any processor that decides before completing stage 0 decides 0. By Lemma 1 and part 3 of Lemma 3, any processor that completes stage 1 and has not already decided, decides 0 at the end of stage 1.

Finally, we show the commit validity condition. Suppose R is failure-free and on-time, and all processors begin with 1. Then every processor completes stage 1. By Lemma 2 and part 3 of Lemma 3, every processor decides 1 at the end of stage 1. □

By our definition of transaction commit protocol, 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.

3.3 Eventual termination

The analysis in this subsection shows that the probability that all processors that complete stage s, decide by stage s, approaches 1 as s approaches infinity. Recall that probabilities are taken over the random information (i.e., the sample space is $\mathcal{F}$), 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 total number of steps taken by p in the run from the beginning until p completes 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 stage s S-message.

The next definition maps a bit to each processor and each stage $s > n$ in a run, such that each stage gets a "new" bit, i.e. a bit independent of the bit assigned to any other stage. 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, $random(R, p, s) = random(R', p, s)$. Note that for a fixed A, I, and s, (A, I, s)-equality is an equivalence relation on $\mathcal{F}$.

In the following three definitions, $s \geq 1$.

Define $v(R, s)$ to be the value of a stage s S-message sent in run R. If no stage s S-message is sent in R, then let $v(R, s) = 0$. By Lemma 4, $v(R, s)$ is well-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 DECIDE(R, s) to be the predicate that each processor that completes stage s has decided by the end of stage s (or earlier) 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. (*Part 1*). The structure of the protocol is such that the random information does not affect which processors send messages to which other processor – 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.

(*Part 2*). This follows from part 1 of this lemma. □

The next lemma states that the value of a stage $s+1$ S-message 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 a processor's stage $s+1$ is not used until the end of that stage, so the same messages are sent in each processor's 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 a simple relationship between MATCH and DECIDE.

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

Proof. Fix $s \geq 1$. At the end of stage s, each processor sets its vote to be either the value received in a stage s S-message or its random value for that stage. MATCH(R, s) means that every processor's random value for stage s is the same as the value sent in any stage s S-message. If MATCH(R, s) is true, then processors set their votes to the same value at the end of stage s, implying that all stage $(s+1, 1)$ messages have the same value

$v \in \{0, 1\}$. By part 3 of Lemma 3, 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 equivalence classes according to the (A, I, s)-equal equivalence relation. 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 $Pr[\text{MATCH}(run(A, I, F), s+1) | F \in C] = 1/2$; if $s \geq n$, then $Pr[\text{MATCH}(run(A, I, F), s+1) | F \in C] = 1/2^n$.*

Proof. (*Part 1*). If $s = 0$, then the result is vacuously true. Suppose $s > 0$. 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).

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

Suppose $s < n$. For F in C, MATCH$(run(A, I, F), s+1)$ is true if and only if $coins(F)[s+1] = v$. Recall that $coins(F)[s+1]$ is equal to either 0 or 1. $\Pr[coins(F)[s+1] = 0 | F \in C] = 1/2$, since C is the set of all elements of $\mathcal{F}$ that are (A, I, s)-equal. Thus $\Pr[\text{MATCH}(run(A, I, F), s+1) | F \in C] = 1/2$.

Suppose $s \geq n$. For F in C, MATCH$(run(A, I, F), s+1)$ is true if and only if $random(run(A, I, F), p, s+1) = v$ for all p. Recall that $random(run(A, I, F), p, s+1)$ is equal to either 0 or 1.

Fix processor p. For any F, the position of $random(run(A, I, F), p, s+1)$ in F depends on whether p completes stage $s+1$ in $run(A, I, F)$ or not. By Lemma 9, either p completes stage $s+1$ in $run(A, I, F)$ for all F in C, or p fails to complete stage $s+1$ in $run(A, I, F)$ for all F in C. If p does not complete stage $s+1$, then $random(run(A, I, F), p, s+1)$ is the second bit of $F(p, s+2)$, obviously a fixed position for all F in C. If p does complete stage $s+1$, then $random(run(A, I, F), p, s+1)$ is the first bit of $F(p, k)$, where $k = index(run(A, I, F), p, s+1)$. By Lemma 9, k is the same for all F in C, so this is also a fixed position for all F in C.

For all distinct p and q and all F in C, the positions in F of $random(run(A, I, F), p, s)$ and $random(run(A, I, F), q, s)$ are distinct.

For any fixed p, $\Pr[random(run(A, I, F), p\, s+1) = 0 | F \in C] = 1/2$, since C is defined by (A, I, s)-equality. Thus $\Pr[\text{MATCH}(run(A, I, F), s+1) | F \in C] = 1/2^n$. □

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, for all $s \geq 1$,*

$$\Pr[\text{MATCH}(run(A, I, F), s)] = \begin{cases} 1/2 & \text{if } s \leq n; \\ 1/2^n & \text{otherwise.} \end{cases}$$

Proof. By part 2 of Lemma 12, since the lemma is true for every equivalence class of $\mathcal{F}$, under the $(A, I, s-1)$-equal relation. □

The next lemma provides a means of calculating the probability of certain compound events. These probabilities will be used in the proofs of Lemmas 15 and 19.

Lemma 14. *Fix adversary A, initial configuration I, and $s \geq 1$. Let $R = run(A, I, F)$ for $F \in \mathcal{F}$ and for all i, $1 \leq i \leq s$, let M_i be either* MATCH(R, i) *or* $\neg$MATCH(R, i). *Then*

$$\Pr[M_1 \wedge \ldots \wedge M_s] = \prod_{i=1}^{s} \Pr[M_i].$$

Proof. The proof is by induction on s. The base case ($s = 1$) is immediate. Suppose the result holds for $s-1$; we show it for s. By the inductive hypothesis, it is sufficient to show $\Pr[M_1 \wedge \ldots \wedge M_s] = \Pr[M_1 \wedge \ldots \wedge M_{s-1}] \cdot \Pr[M_s]$.

By the definition of conditional probability,

$$\Pr[M_1 \wedge \ldots \wedge M_s] = \Pr[M_s | M_1 \wedge \ldots \wedge M_{s-1}] \cdot \Pr[M_1 \wedge \ldots \wedge M_{s-1}].$$

Thus it is enough to show that $\Pr[M_s | M_1 \wedge \ldots \wedge M_{s-1}] = \Pr[M_s]$.

Let X be the set of all $F \in \mathcal{F}$ such that $M_1 \wedge \ldots \wedge M_{s-1}$ is true, where $R = run(A, I, F)$. Partition $\mathcal{F}$ into equivalence classes based on $(A, I, s-1)$-equality. If F is in X, and F and F' are $(A, I, s-1)$-equal, then F' is also in X, by part 1 of Lemma 12. Thus X consists of some finite number of $(A, I, s-1)$-equal equivalence classes; call them $C_1, \ldots, C_k$.

Define q as follows: if $s \leq n$, let $q = 1/2$; if $s > n$ and $M_s =$ MATCH(R, s), let $q = 1/2^n$; and if $s > n$ and $M_s = \neg$MATCH(R, s), let $q = 1 - 1/2^n$. By Lemma 13, $\Pr[M_s] = q$. It remains to show that $\Pr[M_s | M_1 \wedge \ldots \wedge M_{s-1}] = q$. Because $F \in X$ is the same event as $M_1 \wedge \ldots \wedge M_{s-1}$, we have that $\Pr[M_s | M_1 \wedge \ldots \wedge M_{s-1}] = \Pr[M_s | F \in X]$. Now calculate that

$$\Pr[M_s | F \in X] = \sum_{i=1}^{k} \Pr[M_s | F \in C_i] \cdot \Pr[F \in C_i | F \in X]$$

$$= q \cdot \sum_{i=1}^{k} \Pr[F \in C_i | F \in X],$$

by part 2 of Lemma 12

$$= q. \quad \square$$

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

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, $\text{DECIDE}(R, s'+1)$ is true. Since $s'+1 \leq s$, $\text{DECIDE}(R, s)$ is true.

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

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

3.4 Liveness condition

Lemma 15 in the last subsection showed that our protocol terminates in a bounded expected number of stages. Lemmas 16 and 17 in this subsection extend that result to rounds and are used to show the t-nonblocking property in Theorem 18.

Lemma 16. *In any admissible run, 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 admissible run, 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 finish 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 finish 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. There are two cases.

Case 1. All nonfaulty processors complete stage 0 in R. Since R is t-admissible, at most t processors fail in R, and thus every nonfaulty processor completes stage s, for all $s \geq 0$. By Lemmas 16 and 17, $\text{DECIDE}(R, s)$ implies $\text{DONE}(R, 6+2s)$. Lemma 15 gives the result.

Case 2. Some nonfaulty processor p does not complete stage 0 in R. By the code, p is stuck in its request stage. (If a processor ever enters its coins stage, then by $2K$ steps later it enters its vote stage and after at most another $2K$ steps it completes stage 0.) Thus p times out in its request stage after $2K$ steps, which is at most two rounds, and decides 0. Note that p never sends a coins message.

Let q be any nonfaulty processor. If q does not complete stage 0, then the argument in the previous paragraph shows that q decides in at most two rounds. Suppose q does complete stage 0. Since p never sends a coins message, q never receives n coins messages, and thus q times out after at most $4K$ steps, which is at most four rounds, and decides 0. □

3.5 Time complexity

First we show that the expected number of stages of Protocol 1 is less than 4. Then we show that the expected number of rounds is constant. Recall that expectation of complexity measures is defined at the end of Subsect. 2.4.

Lemma 19. *Let X be a random variable giving the least s such that $\text{DECIDE}(R, s)$ is true. Then $E(X, t) < 4$.*

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 s such that $\text{MATCH}(R, s)$ is true. By Lemma 11, $X \leq Y+1$.

$$\begin{aligned}
E(X, t) &\leq 1 + E(Y, t) \\
&= 1 + \sum_{s=0}^{\infty} \Pr[Y > s], \\
&\quad \text{since } Y \text{ is nonnegative integer valued} \\
&= 2 + \sum_{s=1}^{\infty} \Pr\left[\bigwedge_{i=1}^{s} \neg \text{MATCH}(R, i)\right] \\
&= 2 + \sum_{s=1}^{\infty} \left(\prod_{i=1}^{s} \Pr[\neg \text{MATCH}(R, i)]\right), \quad \text{by Lemma 14} \\
&= 2 + \sum_{s=1}^{\infty} \left(\prod_{i=1}^{s} q_i\right) \\
&= 2 + \sum_{s=1}^{n} \left(\prod_{i=1}^{s} q_i\right) + \left(\prod_{i=1}^{n} q_i\right) \cdot \sum_{s=n+1}^{\infty} \left(\prod_{i=n+1}^{s} q_i\right).
\end{aligned}$$

We simplify using specific values for q_i. For $1 \le i \le n$, $q_i = 1/2$, and for $i > n$, $q_i = 1 - 1/2^n$, by Lemma 13.

$$E(X, t) \le 2 + \sum_{s=1}^{n} \frac{1}{2^s} + \frac{1}{2^n} \cdot \sum_{s=n+1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s-n}$$

$$< 2 + 1 + \frac{1}{2^n} \cdot \sum_{s=1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s}$$

$$= 3 + \frac{1}{2^n}(2^n - 1)$$

$$< 4. \quad \square$$

Theorem 20. *In any t-admissible run, 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 $F \in \mathcal{F}$. There are two cases.

Case 1. All nonfaulty processors complete stage 0 in R. As in Case 1 of the proof of Theorem 18, every nonfaulty processor completes stage s, for all $s \ge 0$, and DECIDE(R, s) implies DONE$(R, 6+2s)$. The expected number of stages is four, by Lemma 19. Therefore all nonfaulty processors decide in fourteen expected asynchronous rounds.

Case 2. Some nonfaulty processor p does not complete stage 0 in R. The same argument as in Case 2 of the proof of Theorem 18 shows that every nonfaulty processor decides after at most four asynchronous rounds. $\square$

The proof of the previous theorem shows that every nonfaulty processor decides the fate of the transaction in 14 expected rounds. Recall that expectation is defined with respect to the worst possible adversary, that is, the worst possible scheduling of processor steps and message delays. When the system is well-behaved, our protocol has better performance. In particular, if the coordinator initiates the protocol, the system is synchronous, and there are no late messages or failures, then all the processors decide in $5K$ steps, using $4n^2$ messages.

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 notion of a *guaranteed* message is no longer needed, since atomic broadcast is allowed.

From event $e = (p, M, b)$ applied to configuration C, we compute the *delay* of message m in M (differently from before) as the number of the cycle in which the message is received minus the number of the cycle in which the message is sent. The number of the receiving cycle is obtained from C by looking at the $clock_p$ component of p's state. The number of the sending cycle is obtained from m, which consists of the "sending time" integer tagged onto the raw message; the sending time is the value of the sender's clock variable when m is sent, i.e., the sending cycle number.

An infinite run R is *admissible* if the first configuration is an initial configuration, 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 schedule when a processor fails and select the message delays.

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 by Gray [9]; also analyzed by Halpern and Moses [10]). We partition the processors into two nonempty 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 $turn(C) = turn(D)$, and for all p in S, $state(p, C) = state(p, D)$ and $buff(p, C) \subseteq buff(p, D)$. 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*

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

(2) *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 part 2; part 1 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 all schedules of length $l-1$ or shorter. We show it is true for all schedules of length l. Let $\sigma = \sigma' e$ be a schedule of length l. Since σ' has length $l-1$, by the inductive 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 in e'. 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. Then $e' = e$. We must show that each m in M is in $buff(p, E)$ in order to show that e' is applicable to 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 the first configuration in which m 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 inductive 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 $\tau''(D)$. Since q's transition function is deterministic, m is also sent in $run(D, \tau')$ as a result of event g, and m is in p's buffer in E. □

The next theorem shows that there can be no nonblocking transaction commit protocol if half or more of the processors can fail.

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

Proof. Suppose $n \leq 2t$ and there is a t-nonblocking transaction commit protocol with processors 0 through $n-1$. Let $A = \{0, \ldots, \lfloor n/2 \rfloor - 1\}$ and $B = \{\lfloor n/2 \rfloor, \ldots, n-1\}$. Each of A and B has at most t elements (and at least one element, since $n \geq 2$). The first $\lfloor n/2 \rfloor$ 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 $run(I_{11}, \alpha)$ 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. Figures 1 and 2 illustrate the proof. We show the claim by descending induction on i. Let $C_i = (\pi_1 \ldots \pi_i)(I_{11})$ for $i \geq 0$.

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, other than switching "A" and "B", that are necessary when π_i is an A-semicycle, i.e., when i is odd.)

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 Fig. 1.) By part 2 of Lemma 22, β_1 is applicable to C_{i-1}. Since

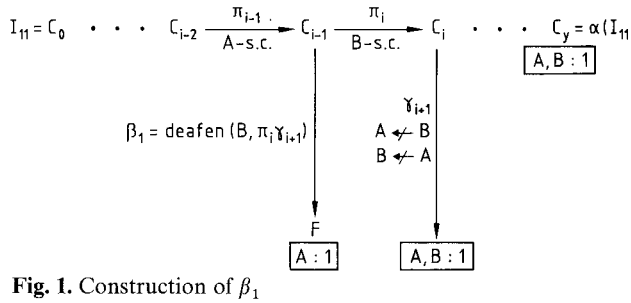

Fig. 1. Construction of β_1

$\beta_1 | A = \pi_i \gamma_{i+1} | A$, 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.

Now we must give a schedule β_2 that causes processors in B to decide 1 without hearing from any processors in A. (See Fig. 2.) 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. Let D be the configuration in $run(I_{11}, \alpha_i)$ 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, i.e., ρ is an A-semicycle followed by a B-semicycle followed by another A-semicycle.

(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 $j-1$, i.e., ρ is an A-semicycle followed by a B-semicycle).

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 $run(I_{11}, \tau)$, i.e., prior to cycle $j-1$, and is in $buff(p, D)$. By part 1 of 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 $\beta_1' = kill(A, \beta_1)$. 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 $run(I_{11}, \tau)$, 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 $run(I_{11}, \tau)$, i.e., before cycle $j-1$. Thus in δ, processors in B receive only messages sent in

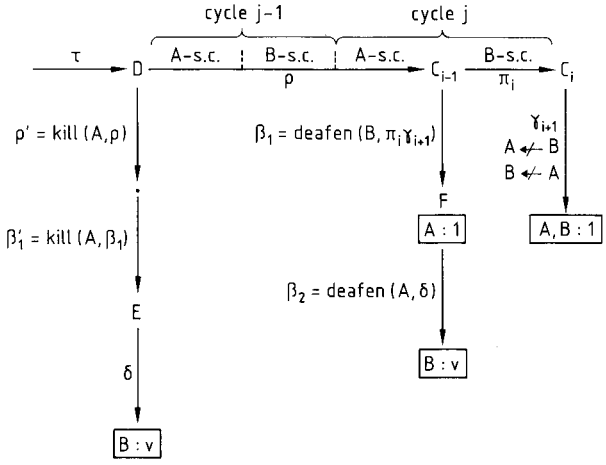

Fig. 2. Construction of β_2

$run(D, \rho' \beta_1' \delta)$. 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 $run(D, \rho' \beta_1')$, i.e., in cycle $j-1$ or later. Lemma 21 implies that q has the same state in corresponding configurations in $run(D, \rho' \beta_1')$ and $run(D, \rho \beta_1)$. Thus q sends the same messages in the two runs, and m is also in $buff(p, F)$. Now we can apply part 2 of 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 part 1 of 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 part 1 of 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 Sect. 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 terminates 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 and any fixed protocol, 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 (with a different adversary) extending from this point that are not deciding. These runs are kept undeciding by delaying the delivery of all messages; they 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 Sect. 4.

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.

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 of the run 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 there is an initial configuration I and a schedule τ applicable to I such that $\tau(I)=C$ and $run(I, \tau\sigma)$ is F-compatible.

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 any 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. (See Fig. 3; in the figure a v in a box below a configuration means that the configuration is $(x, F, \{v\})$-valent.) 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 satisfies the abort validity condition and since F is finite, there is a finite q-free x-slow F-compatible run $run(I_{01}, \sigma)$ such that $\sigma(I_{01})$ has decision value 0, and by the agreement condition, $\sigma(I_{01})$ is $(x, F, \{0\})$-valent.

By part 1 of 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, since I_1 is the desired configuration. 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 in $run(I_1, \tau)$ there must be an event $e=(p, M, b)$ and two adjacent configurations 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. If either configuration is $(x, F, \{0, 1\})$-valent, we have found the desired configuration. Suppose neither is.

Since the protocol is 1-nonblocking, F is finite, no

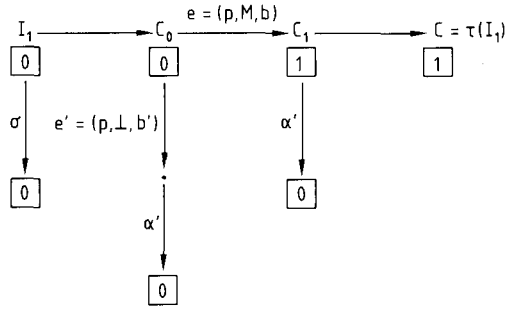

Fig. 3. Demonstrating the existence of an $(x, F, \{0, 1\})$-valent configuration

processor has failed so far, and C_0 is $(x, F, \{0\})$-valent, there is a finite p-free x-slow F-compatible run $run(C_0, \alpha)$ 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.) It is easy to show that 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 $\alpha(C_0)$, they decide 0 in $\alpha'(C_1)$. 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. (For seed F with finite length x, adversary A, and initial configuration I, let $run(A, I, F)$ be the x-cycle run defined by the obvious analogy with the infinite length case in Subsect. 2.4.)

Lemma 25. *Choose any nonnegative integers l and x with $x > l$. 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. 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 $run(A, I_1, F) = run(I_1, \alpha\sigma)$, where α consists of l events and $C = \alpha(I_1)$ is $(x, F, \{0, 1\})$-valent. (See Fig. 4.) 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 for some $v \in \{0, 1\}$. Then there is a configuration D in $run(C, \sigma)$ 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 for some $w \in \{0, 1\}$. M must be the empty set, since no messages are received in $run(C, \sigma)$. 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.

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. It is easy to show that τ is applicable to $e(D)$; τ is also 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). Thus all processors except p decide 1 in $\tau(e(D))$, 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 $\mathscr{R}$ 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 $|\mathscr{R}| = j$. Thus, j is also the number of seeds of length B.

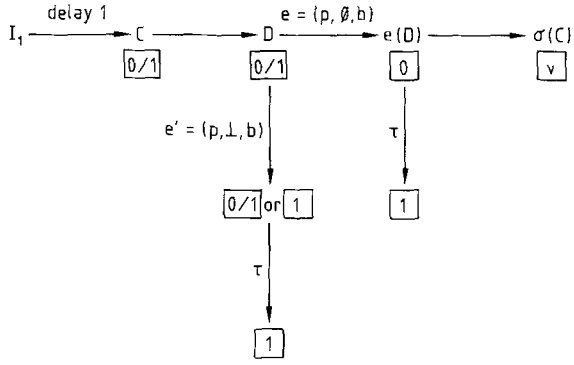

Fig. 4. Demonstrating that $\sigma(C)$ is $(x, F, \{0, 1\})$-valent

Case 1. No run in $\mathscr{R}$ 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 $\mathscr{R}$ that is deciding. Let $\mathscr{C}$ be the set of all configurations in run R, and let $m = |\mathscr{C}|$. Let $\mathscr{S}$ be the collection of all seeds with length jmB that extend the seed of R. $\mathscr{S}$ is finite; in fact, $|\mathscr{S}| = z/j$, where z is the total number of seeds of length jmB.

We will associate each seed in $\mathscr{S}$ with a configuration in $\mathscr{C}$. For each configuration in $\mathscr{C}$, we will associate a specific adversary. The associations will be made in such a way that all runs from a configuration in $\mathscr{C}$, using its particular adversary and any of the associated seeds, is undeciding. The desired adversary is the adversary for that configuration with the most seeds. The extreme length of these undeciding runs will cause the desired expected value to exceed B for this adversary.

For each $C \in \mathscr{C}$, define $S(C)$ to be the set of all $F \in \mathscr{S}$ 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 $F \in \mathscr{S}$ is in $S(C)$ for exactly one configuration C.

Fix C to be a configuration in $\mathscr{C}$ with $|S(C)| \geq \frac{1}{m} \cdot |\mathscr{S}|$. Such a configuration exists by the pigeonhole principle, since $|\mathscr{C}| = m$. Thus, $|S(C)| \geq \frac{1}{jm} \cdot z$, where z is the total number of seeds of length jmB.

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 $\frac{1}{jm}$ of the seeds of length jmB are in $S(C)$. Thus, at least $\frac{1}{jm}$ 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,

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

6 Summary

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 guarantee that each processor terminates in a bounded expected number of its own steps, even if processors run in lockstep synchrony and only one processor can fail.

Acknowledgements. We would like to thank B.H. Liskov, N.A. Lynch, and W.E. Weihl for suggesting this problem to us; A.D. Fekete, Y.O. Moses, and M.R. Tuttle for helpful comments on an early draft; and N.A. Lynch for a very careful reading of a recent draft. We also appreciate the helpful comments of two anonymous referees.

References

1. Ben-Or M: Another advantage of free choice: Completely asynchronous agreement protocols. In: Proc 2nd Annu ACM Symp Principles Distrib Comput 1983, pp 27–30
2. Coan BA, Lundelius J: Transaction commit in a realistic fault model. In: Proc 5th Annu ACM Symp Principles Distrib Comput 1986, pp 40–51
3. Chor B, Merritt M, Shmoys D: Simple constant-time consensus protocols in realistic failure models. J ACM 36:591–614 (1989)
4. Dolev D, Dwork C, Stockmeyer L: On the minimal synchronism needed for distributed consensus. J ACM 34:77–97 (1987)
5. Dwork C, Lynch NA, Stockmeyer L: Consensus in the presence of partial synchrony. J ACM 35:288–323 (1988)
6. Dwork C, Skeen D: The inherent cost of nonblocking commitment. In: Proc 2nd Annu ACM Symp Principles Distrib Comput 1983, pp 1–11
7. Dwork C, Skeen D: Patterns of communication in consensus protocols. In: Proc 3rd Annu ACM Symp Principles Distrib Comput 1984, pp 143–153
8. Fischer MJ, Lynch NA, Paterson MS: Impossibility of distributed consensus with one faulty process. J ACM 32:374–382 (1985)
9. Gray J: Notes on database operating systems. In: Bayer R, Graham RM, Seegmüller G (eds) Operating systems: an advanced course. Lect Notes Comput Sci, vol 60. Springer, Berlin Heidelberg New York 1978, pp 393–481
10. Halpern JY, Moses YO: Knowledge and common knowledge in a distributed environment. In: Proc 3rd Annu ACM Symp Principles Distrib Comput 1984, pp 50–61 (revised as of Jan. 1986 as IBM-RJ-4421)
11. Rabin MO: Randomized Byzantine generals. In: Proc 24th Annu IEEE Symp Found Comput Sci 1983, pp 403–409
12. Skeen D: Crash recovery in a distributed database system. Ph.D. dissertation, University of California, Berkeley 1982 (available as UCB/ERL M82/45)