The Inherent Cost of Nonblocking Commitment†

Cynthia Dwork
Dale Skeen

Computer Science Department
Cornell University
Ithaca, New York 14853

Abstract

A commitment protocol orchestrates the execution of a distributed transaction, allowing each participant to "vote" on the transaction and then applying a prespecified rule to decide the outcome (commit or abort). A *nonblocking* commitment protocol is able to correctly terminate a transaction at all operational participants in the presence of any number of benign processor failures. Herein, we derive strong lower bounds for both nonblocking protocols and their less fault-tolerant blocking counterparts. Results on message complexity are both surprising and encouraging: the message complexities of the two classes of protocols are identical. Results on time complexity were less encouraging: nonblocking protocols are approximately 50% more expensive. However, we show how to overlap nonblocking executions of interfering transactions and thereby reduce their extra cost.

†This work was supported in part by the NSF grant MCS81-01220.

1. Introduction

A *distributed transaction* is an atomic action spanning multiple processors; either all or none of its effects persist. The transaction notion is fundamental in fault-tolerant systems – useful both in the conceptualization and the realization of such systems. Historically transactions have been associated with database systems; however, the notion has broad applicability.

In execution, a distributed transaction (henceforth, a "transaction") is decomposed into *subtransactions*, each of which is executed by a single processor. A *commitment protocol* orchestrates the execution of the subtransactions among the participating processors, establishing the all or nothing appearance of the transaction. A transaction is only as fault-tolerant as the commit protocol coordinating its execution.

An extremely important class of fault-tolerant commitment protocols is the *nonblocking protocols*. A *nonblocking protocol* can correctly terminate a transaction as long as processor failures are not malicious and one of the participating processors remains operational. Hence, such protocols never "block" (suspend execution) because of processor failures, and in this sense, they are maximally tolerant of benign processor failures.

In spite of their increased fault-tolerance, nonblocking protocols are often not used because of their expense: all known nonblocking protocols are approximately 50% more costly than their blocking counterparts. The same overhead is found whether the cost metric is message counts or tandem message delays. Message counts are a rough measure of the network bandwidth required to support the protocols; whereas tandem message delays are often a large component in the execution time.

Our goal is to study the inherent cost difference between blocking and nonblocking protocols, with *messages* and *time* as metrics. Specifically, we study the cost of a "best-case" instance of a protocol; the "best-case" occurring when none of the possible failures materialize. Failure-free performance issues are important in practice; when failures are infrequent, which is the case for most environments, failure-free performance is a good indicator of expected performance.

Our results on message complexity are positive. While blocking protocols with best-case message complexity $2(n-1)$ were known, nonblocking protocols were generally thought to require about $3(n-1)$ messages. We were at first frustrated by our attempts to prove this disparity inherent in the differences between the two classes of protocols. Our continued inability to close the gap induced us to reexamine extant protocols. This led to a surprising discovery: a new *nonblocking* protocol with best-case message complexity $2(n-1)$. Then, having convinced ourselves that the $2(n-1)$ conjectured lower bound for either class of protocols was no longer obvious, we proceeded to prove it for both.

The results for time are less encouraging: in the absence of failures the fastest nonblocking protocol requires roughly twice as much time as the fastest blocking protocol. However, this negative result is partially compensated by an interesting observation on the implementation of nonblocking protocols: nonblocking protocols exhibiting a certain property can allow more concurrency among conflicting transactions than previously thought. This increased concurrency attenuates the performance degradation expected in transaction systems using nonblocking protocols.

2. Background

2.1 The Environment

We make the following assumptions concerning the network:

(1) the network is fully connected,

(2) messages between operational processors are correctly delivered,

(3) spurious messages are not generated,

(4) the maximum time required for a processor p to send a message and receive a reply is $2\Delta_p$, for some Δ_p measured on the processor's local clock.

Δ_p represents the maximum end-to-end message delay[1], and includes the physical transmission time plus the message processing time incurred by both the sender and receiver.

These are strong assumptions; nonetheless, they are frequently assumed for many applications. Relaxing (3) requires solving a variant of the Byzantine General's problem at considerably more expense ([D], [DFFLS], [DR], [DoS], [LFF], [LSP], [PSL]). Relaxing (4) makes the problem unsolvable for even the two-processor case [FLP]. Implementable networks can approximate (4) to an arbitrarily high degree of certainty with an appropriate choice of Δ_p for each processor p.

A system satisfying (3) and (4) can be modeled as a synchronous system with a clock cycle time of $\Delta = \max_p \{\Delta_p\} + \delta$, where δ is a function of the maximum rate of drift between the processors' clocks. This observation follows directly from the results in [LM]. For simplicity we will assume a completely synchronous system. In one *step* each processor can:

- receive an arbitrary number of messages (at most 1 from any processor);
- change state;
- send at most k messages;

We take k to be 1; however, the particular choice of k is irrelevant provided the sending of multiple messages is not assumed to be atomic. We assume that failed processors do not recover during the execution of a transaction.

2.2 Commitment Protocols

Transactions are decomposed into subtransactions, which are then distributed to participating processors for execution. Commitment of a subtransaction is rarely automatic, rather, each processor is given the opportunity to vote ("accept" or "reject") on its subtransaction. Rejection may occur for a variety of reasons, for example, the subtransaction may deadlock with other tasks, or a requested item may simply not be available. Also, a processor may fail before voting, and this is normally interpreted as an implicit "reject."

[1] An *end-to-end message delay* is the elapsed time between the sending of a message by an application on one processor and the receipt of that message by the application on another processor.

A *commit rule* governs when a transaction may be ommitted, and it is a function of the votes. A fre- uently used rule, assumed herein, is *unanimity*: any implicit or explicit) vote to reject causes abortion.

Subtransaction processing proceeds in well-defined teps, which we model as states of a finite state nachine. Processing can terminate in one of two *final tates, abort* or *commit*, which have the obvious emantics. Final states are terminal: no transitions manate from them. Commitment and abortion then re **irreversible**.

A transaction is in an *inconsistent state* whenever ome subtransactions are committed while others are borted. Partial correctness for a commitment proto- ol is captured in the following two rules:

C1. It preserves consistency.

C2. It commits a transaction only if the commit rule is satisfied.

'otal correctness requires that transactions satisfying he commit rule actually be committed, in the absence f failures.

The concepts of *committable* and *noncommittable* tates are crucial to the understanding of nonblocking rotocols. The occupancy of a committable state nplies the satisfaction of the commit rule. The com- iit state is obviously a committable state; whereas he abort state is not. In addition to partial correct- ess, *nonblocking* protocols satisfy the rules given elow [S82]. p is an arbitrary processor.

N1. p commits only if every nonfailed processor occupies a committable state.

N2. p aborts only if every nonfailed processor occupies a noncommittable state.

'he committable and noncommittable states of a pro- :ssor are specified *a priori* in the protocol.

. Message Complexity

We begin by proving the lower bound.

Lemma 1 (informal version): Let P be a (blocking or nonblocking) protocol satisfying the consistency condition $C1$ and the commit rule $C2$, and let p be an arbitrary processor in P. If I is a failure-free instance of P resulting in commitment of a transaction, then for all processors $q \neq p$ there must be a path of messages from p to q. In other words, a certain amount of information must be transmitted explicitly from p to each other processor.

Let I require time t. We construct the *message graph* $G = (V,E)$, corresponding to instance I. The vertex set, V, is a grid of $t + 1$ columns and n rows. A vertex is specified by a pair of grid coordinates. Column 0 represents the processors before the vote, and in general column i represents the processors at time i. We let $E = E_m \bigcup E_r$, where E_m, the set of *message* edges, represents the flow of information between pairs of distinct processors, and the *row* edges, E_r, represent the (trivial) flow of information from processors to themselves. More formally,

$$E_m = \{<(p,i),(q,i+1)> \mid p \text{ writes to } q \text{ at time } i\},$$
$$E_r = \{<(p,i),(p,i+1)> \forall p, \forall i: 0 \leq i < t\}.$$

The edges are directed.

Definition: An (n,m)-network is a directed, acyclic graph with n inputs (vertices of indegree 0) and m outputs (vertices of outdegree 0).

Definition: An *n-distributor* is a $(1,n)$-network in which there is a path from the input to each of the n outputs.

Fix a processor p satisfying the conditions of the lemma, and let $G(p)$ be the directed acyclic subgraph of G rooted at $(p,0)$.

Claim 1 (Formal version of lemma 1): $G(p)$ is an n-distributor.

Proof: We define the coloring function, $C: V \rightarrow \{red, blue\}$ by

$$C((r,i)) = \begin{cases} red & \text{if } (r,i) \in G(p) \\ blue & \text{if } (r,i) \notin G(p) \end{cases}$$

The coloring is extended to edges by coloring each edge according to the color of its source. Edges from red vertices are red, edges from blue vertices are blue. The red subgraph of G is exactly $G(p)$. A message represented by a red (blue) edge is called a *red* (*blue*) *message*. If (r,i) is in $G(p)$ then every (r,j), $i \leq j \leq t$, is also in $G(p)$. Thus, row p is completely red. Every other row of the grid is initially blue, and remains blue until it is reached by a red edge. From that point on the row is red.

Let us assume that $G(p)$ is not an n-distributor. Then there is a nonempty set of

processors, Q, such that for every q in Q, $(q,t) \notin G(p)$. Then for each $q \in Q$, every edge incident on row q is blue, so the entire row is blue.

We now construct a run in which p fails before voting (an implicit "reject"), but the processors in Q receive exactly the messages they receive in instance I. For each processor r, let $red(r)$ denote the least i for which (r,i) is red. We say processor r is blue until $red(r)$, after which it is red. Consider the run in which every processor r fails immediately after it sends its last blue message. At any instant, the state of a blue processor in the bad run is identical to its state at the corresponding instant of I, so it sends exactly the same messages in both runs. Whether a processor has failed (in the bad run) or turned red (in I), it does not write to processors in Q. Thus, a nonfailed processor cannot distinguish the good run from the bad on the basis of messages received or messages not received due to failure of the sender. In particular, the processors in Q can never distinguish the good run from the bad. By assumption, all processors commit the transaction in instance I. The processors in Q must therefore commit in the bad run, even though the commit rule is not satisfied, violating *C2*. □

Definition: An (a,b)-distributor is an (a,b)-network in which each input induces a b-distributor.

Corollary 1: Let P be any protocol satisfying the conditions of consensus and unilateral abort, and let I be a failure-free instance of the protocol resulting in commitment. Then the message graph corresponding to I is an (n,n)-distributor.

Proof: Immediate from n applications of lemma 1. □

Notation: Let $|S|$ denote the cardinality of S, for any set S.

Corollary 2: $|E_m \cap G(p)| \geq n - 1$.

Proof: By lemma 1, $(r,t) \in G(p)$, for all processors r. If $r \neq p$ then there is at least one edge in $G(p)$ incident on row r and originating in some row $q \neq r$. E_m is precisely the set of messages between distinct rows, so $|E_m \cap G(p)|$ is at least $|\{r \mid r \neq p\}| = n - 1$. □

Lemma 2: Let i be in the range $1 \leq i \leq n$, and let S be a set of i distinguished processors, without loss of generality, $S = \{p_1, \ldots, p_i\}$. Let M be the set

$$M = \bigcup_{p_j \in S} (E_m \cap G(p_j)).$$

Then $|M| \geq n + i - 2$.

Proof: The proof is by induction on i, the cardinality of S. For the basis, $i = 1$, the proof is immediate from corollary 2.

For $i \geq 2$, assume the lemma holds for $i - 1$ and let S be as in the statement of the lemma. We assume, for the sake of contradiction, that $|M| \leq n + i - 3$.

It is not hard to show the existence of a processor $p \in S$, and message edge $e \in (G(p) \cap E_m)$ such that $\forall p' \in S - \{p\}$: $e \notin G(p')$. That is, p sends the message corresponding to e before p is reached by any of the other processors in S. Fix any such p and e, and let $S' = S - \{p\}$. Let M' be defined analogously to M:

$$M' = \bigcup_{p_j \in S'} (E_m \cap G(p_j)).$$

Then $e \notin M'$. By the induction hypothesis $|M'| \geq n + i - 3$. Since M' is properly contained in M, $|M'| \leq |M| - 1$. By the assumption on $|M|$, $|M'|$ is at most $n + i - 4$, a contradiction. □

Theorem 1: Any commitment protocol satisfying the consistency condition *C1* and the commit rule *C2* requires at least $2(n-1)$ messages to commit a transaction in the absence of processor failures.

Proof: Let $i = n$ in the proof of lemma 2. □

Theorem 1 provides a lower bound for both bloc ing and nonblocking protocols. While blocking prot cols achieving the lower bound are well-known, it h been previously conjectured that this bound was t

weak for nonblocking protocols. This, however, is not the case.

Theorem 2: There exists a nonblocking commitment protocol requiring exactly $2n - 2$ messages in the presence of no processor failures. The time required is $2n + 1$, regardless of the number of processor failures.

Proof:

Protocol A, which appears in the appendix, achieves the $2n-2$ message bound. It has three phases: voting, reporting, and confirmation (it assumes that subtransactions have already been distributed to processors). In the voting phase (step 0) processors send their vote to a distinguished processor, p_0, which does nothing until step 1. During step 1 the distinguished processor receives the votes, casts its own vote, computes the result, and begins the reporting phase, sending the result to processor p_1. If p_0 should remain operational throughout this phase, it sends the result of the vote to each p_i at step i. Upon receipt of this message (at step $i+1$), p_i enters the abort state or a committable state (not a commit state), according to result of the vote.

If p_0 should fail during the reporting phase, and p_i ($i \geq 1$) is operational at step $i+1$, then p_i will become aware of the failure because its anticipated message will not arrive. If $i > 1$ then p_i requests help from each p_j, for $1 \leq j < i$, writing to p_j at step $j+i$, until it receives a response. If, in addition, all these processors fail before sending to p_i, then p_i is never informed of the result of the vote, and consequently enters an abort state. If p_0 fails before sending to p_1, then p_1 enters an abort state at step 2.

If p_j ever receives a request for help from p_i, $i > j$, it does so at time $j+i+1$. Whether or not p_j receives a distress message from p_i, if p_j has not failed, then at step $j+i+2$ it knows that p_i has either failed or knows the result of the vote. If p_j is committable, then, since $i \leq n-1$, p_j may enter a commit state at the end of step $j+n+1$. Further, in claim 3 we prove that p_j receives no requests for help before step $2j + 2$; thus, p_j is active in the confirmation phase of the algorithm during steps s, for $2j+2 \leq s \leq n+j+1$.

Claim 2: If p_i has not failed by the end of step $2i + 1$ then it has decided (*abort* or *commit*).

Proof: Immediate from the algorithm. □

Claim 3: p_i receives no request for help before step $2i + 2$.

Proof: A processor p_j only requests help from processors p_k for $k < j$. Let j and i, $j > i$, be fixed. If p_j requests help from p_i it does so at step $i + j$, and p_i does not receive the message until step $i + j + 1$. Since $j > i$ this is at least $2i + 2$. □

Once decided, processors remain that way or fail, and failed processors do not recover during execution of the transaction. This, together with the order in which processors are polled and the time at which a given processor begins polling, guarantees that if p_j is undecided so are all p_k, $k > j$, for which $vote(k) = commit$. This justifies requesting help only from processors with smaller indices.

Claim 4: The algorithm runs in $2n + 1$ steps.

Proof: Each p_i acts according to decision(i) at step $n + i + 1$. When $i = n-1$ this is $2n$. Since the algorithm begins with step 0 we have a total of $2n + 1$ steps. □

Claim 5: In the absence of processor failures the algorithm requires exactly $2(n-1)$ messages to commit a transaction.

Proof: During step 0 each of $n-1$ processors p_i sends its vote to p_0. By assumption p_0 does not fail, hence at step i it reports the result of the vote to processor p_i, which receives the message at step $i+1$, as expected. Thus each processor sends exactly one message except p_0, which sends $n-1$, for a total of $2(n-1)$ messages. □

This completes the proof of theorem 2. □

Our improvement over the conjectured lower bound is in the third phase, wherein explicit confirmation messages are omitted *at no cost in time*. This technique is used in an identical fashion in [CD], and a similar idea appears in [L]. Although the initial

phase of distributing the subtransactions was not counted, this can be achieved in conjunction with the voting phase *with no extra messages.*

4. Time Complexity

Information can be transmitted by a "nonmessage": in the absence of failures, the lack of a distress message within a bounded time guarantees the satisfaction of the nonblocking rules $N1$ and $N2$. Therefore, the message bandwidth of nonblocking protocols can be reduced to that of blocking protocols. A complementary question is whether the execution time of nonblocking protocols can be reduced to that of blocking protocols, possibly at the expense of more messages?

The answer is negative. In fact, the fastest nonblocking protocol requires roughly twice as much time as the fastest blocking protocol.

Lemma 3: Any commitment protocol of size n requires time $\log_2 n$.

Proof: By lemma 1 each processor must explicitly reach each other processor, and the number reached at most doubles at each step. The result follows by an easy induction on time. □

Theorem 3: Any nonblocking commitment protocol of size n requires time at least $2\log n - 3\log\log n - O(1)$.

Proof: For simplicity, if the commit rule is satisfied, then a nonfailed processor p is committable at step s if and only if for all processors q, $(p,r) \in G(q)$, i.e., if and only if it has been reached by all processors.

Let P be a time-optimal nonblocking protocol and I an instance of P resulting in commitment and requiring time t (we can show $t < 2\log_2 n$). There exists a step in which at least n/t processors become committable. Let step r be such a step, M the set of processors becoming committable at r, and S the processors not in M. Let $N = S \bigcup M$.

Claim: If $|M| \geq 3$, then the elements of M are not in commit states at the end of step r.

Justification: Let $x,z \in M$. Since z becomes committable at r it must receive at least one message sent at $r-1$. At step r, the only processors that know the message was sent to z at $r-1$ are the sender and the receiver. Thus, x can know at r that the message was sent only if x sent it. Since x can send to at most one processor during step $r-1$, there is at most one processor (other than x) in M known by x to be failed or committable at the end of step r and before step $r+1$. □

Let k steps suffice to move the elements of N to commit states. At most $n(k+1)$ messages can be received in the $k+1$ steps $r,r+1,\ldots,r+k$ of I. Suppose, for the sake of argument, that all these messages are received by processors in M. Then there exists some processor, $p \in M$, such that p receives at most $n(k+1)/|M|$ messages during these $k+1$ steps. Let $c = n/|M|$ and let $d = c(k+1)$. Then p receives at most d messages in steps r through $r+k$.

Let G be the message graph corresponding to instance I, as in the proof of lemma 1. For all vertices $v \in G$ let $D(v)$ denote the directed, acyclic subgraph of G rooted at v. We examine the subgraph of G induced by columns $r-1$ through t. Let $G(p) = D((p,r))$. Let $H(p)$ denote the subgraph induced by the processors sending to p at or after step $r-1$ of I. Thus,

$$(x,i) \in H(p) \Leftrightarrow \begin{cases} ((x,i-1),(p,i)) \in G \ \wedge \ i \geq r \ \text{ or} \\ (x,i) \in D(v) \ \text{ for some } \ v \in H(p) \end{cases}$$

From I we will construct a run J, in which some arbitrary $p \in M$ remains operational but does not become committable at r. Let $F(p)$ be the subgraph of G induced by processors sent to by p in J. That is,

$$(x,i) \in F(p) \Leftrightarrow \begin{cases} p \text{ sends to } x \text{ at } i\text{-}1 \text{ in } J \ \wedge \ i > r \\ \text{or } (x,i) \in D(v) \ \text{ for some } v \in F(p) \end{cases}$$

Note that although defined by the behavior of p in J, $F(p)$ is a subgraph of G, which corresponds to instance I.

Notation: Let $FGH(p)$ denote the union of $F(p)$, $G(p)$, and $H(p)$. That is, $FGH(p) = F(p) \bigcup G(p) \bigcup H(p)$.

Specification of J: For all $x \neq p$, x fails just prior to step $\mu i \{(x,i) \in FGH(p)\}$, denoted $fail(x)$.

Claim: If $(x,i) \notin FGH(p)$ then the state of x at i in I is the same as the state of x at i in J.

Justification: The initial state of x and the messages sent and received by x in steps 0 through i are identical in the two runs. □

Corollary: For all processors x, $(x,t) \in FGH(p)$ in run J.

Proof: Suppose not. Let q be any processor such that $(q,t) \notin FGH(p)$. In run I q commits its subtransaction by step t. By the claim, if $(q,t) \notin FGH(p)$ then q still cannot distinguish the two runs at t. Thus q commits in J as well, violating rule $N1$. □

Let $\alpha = \max_z \{fail(z)\} - r$ and let z fail at $r + \alpha$. Then z cannot distinguish I from J before $r + \alpha$, so z cannot commit in I before time $r + \alpha$, or it will do so erroneously in J. Therefore, any failure-free run resulting in commitment requires time at least $r + \alpha$. Since $t = r + k$ we have $k \geq \alpha$.

Let the function $f(i)$ bound the size of $FGH(p)$ at time $r+i$. At the end of step r runs I and J differ only in the states of p and processors sending to p at $r-1$. Since at most d processors send to p at or after step $r-1$ of I, at most d can possibly write to p at step $r-1$. Thus, $f(0) = d + 1$.

Each of these processors can send at most one message during step r of I. Further, p can send exactly one explicit message in step r of J. Together the processors in $FGH(p)$ can send a total of $f(0)$ implicit messages and 1 explicit message in step r of J, none of which are received until step $r+1$. Thus, at the end of step $r+1$ the two runs differ in the states of at most $2f(0) + 1$ processors, so $f(1) = 2(d+2) - 1$. In general,

$$\begin{aligned} f(i) &= 2f(i-1) + 1 \\ &= 2(2^{i-1}(d+2) - 1) \\ &= 2^i(d+2) - 1. \end{aligned}$$

The least α such that $N = FGH(p)$ at the end of step $r + \alpha$ in J satisfies:

$$\begin{aligned} & 2^\alpha(d+2) - 1 \geq n \\ & \Rightarrow 2^\alpha(d+2) > n \\ & \Rightarrow 2^\alpha > n/(c(k+1)+2) \\ & \Rightarrow 2^\alpha > (n/c)/(k+1+2/c) \\ & \Rightarrow \alpha > \log(|M|) - \log(k+1+2/c). \end{aligned}$$

But $c > 1$ and $|M| \geq n/t$, so $\alpha \geq \log(n/t) - \log(k + O(1))$, whence,

$$(1) \quad \alpha \sim \log(n/t) - \log\log n - O(1),$$

since $k < t \leq 2\log n$.

How large is r? Every processor must explicitly reach every committable processor, and these number at least n/t by the end of step r of I. Thus, $r \geq \log_2(n/t)$, and

$$(2) \quad \log(n/t) + \alpha \leq r + \alpha \leq r + k = t.$$

From the bound on t, and using (1) as an approximation to α, we rewrite (2) to obtain

$$2\log n - 3\log\log n - O(1) \leq t.$$

□

5. An Observation on Efficient Implementation

Nonblocking protocols inherent incur more end-to-end message delays, and these delays are often significant when compared to local processing time. Since transactions lock the objects they touch (or reduce their accessibility by similar means), nonblocking transactions render objects inaccessible for a longer period of time. This, not the increase in message bandwidth or in local processing costs, is the real cost of tolerating arbitrary processor failures. On high contention items, reduced availability translates to reduced throughput.

It is the time during which accessibility of shared objects is restricted that is important, not the elapsed time of transaction execution. Historically, these two time intervals have been co-extensive; however, this may not be necessary. If the interval of restricted accessibility is reduced, the execution of conflicting transactions can be partially overlapped, thus ameliorating some of the most significant expense of fault-tolerance. A closer examination of Protocol A, especially a better understanding of its intermediate states, suggests that such a reduction can be achieved.

Figure 1 depicts the subtransaction states occupied for both types of protocols (blocking and nonblocking). States are shown in chronological order (left to right) for a committed transaction. (While not every subtransaction need occupy all of the illustrated states, at least one must.) The nonblocking subtransaction contains an additional state, the intermediate *committable* state, whose existence is dictated by the nonblocking rules $N1$ and $N2$. For Protocol A, the length of the committable state is exactly $n\Delta$.

Protocol A exhibits an important property: once the reporting phase begins, a committable transaction is committed so long as a single processor in the committable state remains operational. This can be restated as a local condition for each processor: once committable, the processor will eventually commit the transaction or fail. Hence, for each processor p, the predicate $committable(p) \vee failed(p)$ is montonic. Protocols exhibiting this monotonicity are known as *progressive* protocols [S81]. The probability of aborting a transaction when using a progressive protocol decreases rapidly as more and more processors are made aware of its committability. By studying the communication topology of such a protocol, the probability function for abortion can be estimated and, more importantly, the instant that abortion becomes impossible can be determined. This idea was exploited in eliminating the final "commit" messages in the above protocol.

Consider now two subtransactions executing on the same processor and in competition for a shared data object. Normally, the subtransactions would be executed serially; the second transaction waits for the first to commit before starting, as shown in Figure 2a. Alternatively, subtransaction processing could be partially overlapped, by employing the progressive strategy. The second subtransaction reads the first's committable (but not committed) results and then delays voting until commitment of the read results is certain ($n\Delta$ in Protocol A). The situation is depicted in Figure 2b.

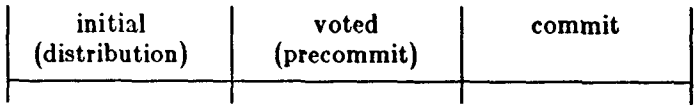
The ordering of events ensures:

(1) transition of the first subtransaction into the committable state precedes initiation of the second transaction.

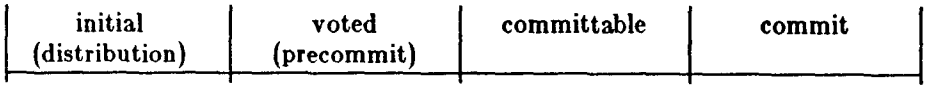
(2) commitment of the first precedes voting of the second.

Referring again to Figure 2b, the second subtransaction must be rejected if the first ultimately aborts. However, once in a committable state, the first aborts only if the host processor fails, and in this case, the second subtransaction would independently and implicitly be rejected by the mechanics of transaction processing. Hence subtransaction abortions may be correlated but never causal: the second transaction is never rejected solely because it read uncommitted results. Therefore, throughput can only increase by adopting this scheme.

We have discussed only subtransactions on a single processor. This technique can sometimes be extended to subtransactions executing on different processors.



a. A blocking subtransaction.



b. A nonblocking subtransaction.

Figure 1. Subtransaction states when blocking and nonblocking protocols are used. Popular alternative names for states are given parenthetically. (State occupancy times, as indicated by the length of intervals, are not drawn to scale.)

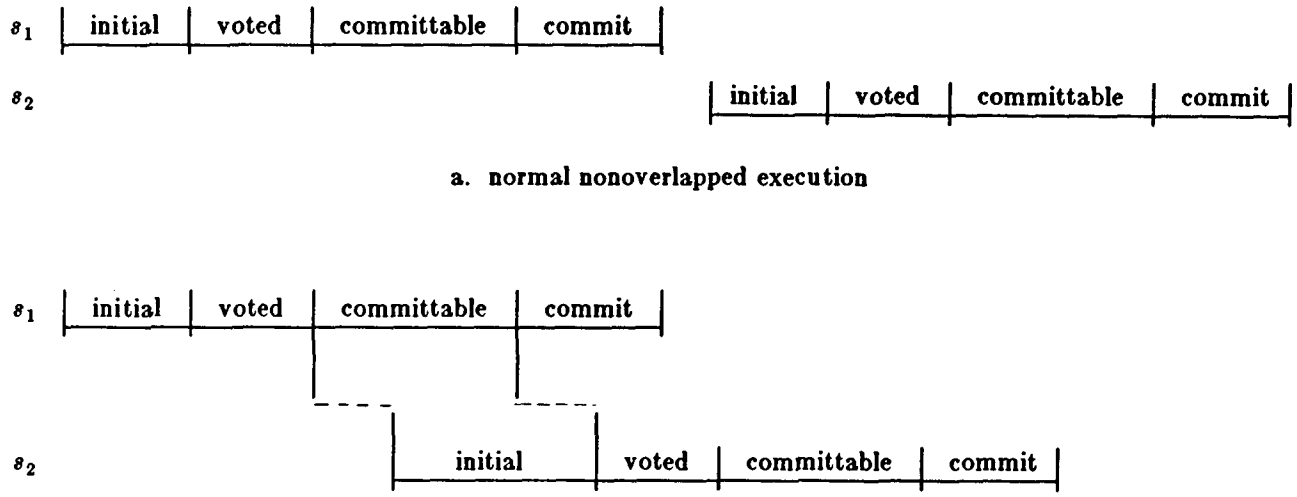


Figure 2.
Processing the conflicting subtransactions s_1 and s_2.

he major network requirement for the extension is at processor failures be detected and failure information be propagated in a timely fashion. ecifically, the elapsed time between the failure of ocessor i and the receipt of the failure notification processor j must be bounded. In the above example, if the two subtransactions were at different processors, then the second would have to delay voting an ditional F time units, where F is the maximum lay for the second host to learn of the failure of the st. For sizable subtransactions, the initial phase is mputationally intensive and may well exceed the quired delay time.

Discussion

We have shown that there is no fundamental ference in message complexity between blocking d nonblocking protocols. This surprising result has legated a contrary "folk theorem" to its proper ace in mythology. Moreover, the proof is in the m of interesting new nonblocking protocol.

On the other hand, there appears to be a fundaental difference in the execution times required by different protocols. This is disappointing since it anifests itself as a decrease in throughput of transacns systems using nonblocking protocols. The extent the degradation is unknown since no systems using h protocols have been measured, but it could be bstantial.

We have proposed, however, a scheme for introducing more parallelism into nonblocking systems; thus, reducing the performance penalties inherent in these systems. Our scheme was motivated by the discovery of a new protocol and a better understanding of its formal properties. Although it allows transactions to read uncommitted data, it does not exhibit the undesirable properties normally found in such schemes, specificly, (1) it does not require additional message traffic between dependent transactions, and (2) it does not cascade aborts, in fact, transactions are never aborted solely because they read uncommitted data. To achieve these properties, properties of progressive protocols are exploited.

7. References

[CD] S. Cook and C. Dwork, "Bounds on the Time for Parallel RAM's to Compute Simple Functions," *Proc. 14th ACM Symposium on the Theory of Computing*, 1982, pp. 231-233.

[D] D. Dolev, "The Byzantine Generals Strike Again," J. Of *Algorithms*, vol. 3, no. 1, 1982.

[DFFLS] D. Dolev, M. Fischer, R. Fowler, N. Lynch, and R. Strong, "Efficient Algorithms for multiple Processor Agreement," *Proc. 14th ACM Symposium on the Theory of Computing*, 1982.

[DR] D. Dolev and R. Reischuk, "Bounds on Information Exchange for Byzantine Agreement," *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing*, 1982.

[DoS] D. Dolev and H.R. Strong, "Requirements for Agreement in a Distributed System," *Proc. 2nd International Symposium on Distributed Databases*, 1982.

[FLP] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *2nd Annual ACM Symposium on the Principles of Database Systems*, 1983, pp. 1-7.

[L] L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," to appear in *ACM Trans. on Programming Languages and Systems.*

[LM] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *Technical Report Op.60, Computer Science Laboratory, SRI International*, 1981.

[LSP] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. on Programming Languages and Systems*, vol. 4, no. 3, 1982.

[LFF] N. Lynch, M. Fischer, and R. Fowler, "A Simple and Efficient Byzantine Generals Algorithm," *Proc. 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1982.

[PSL] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *JACM, vol. 27, no. 2*, 1980, pp. 228-234.

[S81] D. Skeen, "A Decentralized Termination Protocol," *Proc. 1st IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1981.

[S82] D. Skeen, "Crash Recovery in a Distributed Database System," Technical Report UCB/BRL M82/45, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1982.

APPENDIX

B. Protocol A.

Let p_0 be a distinguished processor and let tl other processors be numbered from 1 to $n-1$. Eac p_i, $i \geq 2$, behaves as follows:

```
if (step = 0) →
    if vote(i) = abort → {unilateral abort}
        decided(i), decision(i) := true, abort;
    [] vote(i) = commit → decision(i) := fa
    fi
    Send vote(i) to p0;
[] 1 < step ≤ i → skip;
[] i < step ≤ n + i →
    if (~decided(i)) →
        if (decision message received) →
            decided(i) := true;
            decision(i) := message;
        [] (no decision message received) →
            j := step - i;
            if (j < i) →
                write "help" to pj;
            [] (j = i) → skip;
            [] (j = i + 1) →
                decided(i) := true;
                decision(i) := abort;
            fi
        fi
    [] (decided(i) ∧ "help" message received)
        {message is from p_{step-i-1}}
        send decision(i) to p_{step-i-1}
    [] (decided(i) ∧ no "help" message receiv
        {p_{step-i-1} failed or decided} skip;
    fi
[] step = n+i+1 → act according to decisio
[] step > n+i+1 → skip
fi
```

The algorithm for p_1 is similar, but if p_1 does not h from p_0 during step 2 it knows the result of the v

has been lost, and can decide to abort at that time.

The algorithm for p_0 is simple:

```
if   step = 0 → skip;
[]   step = 1 →
     if   (n-1 commit votes are received
               ∧ vote(0) = commit) →
                    decision(0) := commit;
          [] (fewer than n-1 commit votes received
               ∨ vote(0) = abort) →
                    decision(0) := abort;
     fi
fi
if   1 ≤ step < n  → write decision(0) to p_step;
[]   step ≥ n  → skip;
fi
```