THE INHERENT COST OF NONBLOCKING COMMITMENT

Cynthia Dwork
Dale Skeen

TR 82-538
May 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work was supported in part by the NSF Grant MCS81-01220.
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 pre-specified rule to decide the outcome (commit or abort). A nonblocking 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 non-blocking 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 non-blocking executions of interfering transactions and thereby reduce their extra cost.

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 commitment 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, non-blocking 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.

†This work was supported in part by the NSF grant MCS81-01220.
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 \(\Delta_p\) measured on the processor’s local clock.

\(\Delta_p\) 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 \(\Delta_p\) for each processor \(p\).

A system satisfying (3) and (4) can be modeled as a synchronous system with a clock cycle time of \(\Delta = \max \{\Delta_p\} + \delta\), where \(\delta\) 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 committed, and it is a function of the votes. A frequently used rule, assumed herein, is *unanimity*: any (implicit or explicit) vote to reject causes abortion.

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

A transaction is in an *inconsistent state* whenever some subtransactions are committed while others are aborted. Partial correctness for a commitment protocol is captured in the following two rules:

*C1.* It preserves consistency.

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

Total correctness requires that transactions satisfying the commit rule actually be committed, in the absence of failures.

The concepts of *committable* and *noncommittable* states are crucial to the understanding of nonblocking protocols. The occupancy of a committable state implies the satisfaction of the commit rule. The commit state is obviously a committable state; whereas the abort state is not. In addition to partial correctness, *nonblocking* protocols satisfy the rules given below [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.

The committable and noncommittable states of a processor are specified *a priori* in the protocol.

3. 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* ≠ *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*<sub>m</sub> ∪ *E*<sub>r</sub>, where *E*<sub>m</sub>, the set of *message* edges, represents the flow of information between pairs of distinct processors, and the *row* edges, *E*<sub>r</sub>, represent the (trivial) flow of information from processors to themselves. More formally,

*E*<sub>m</sub> = {(*(p,i),(q,i+1)*) | *p* writes to *q* at time *i*},

*E*<sub>r</sub> = {<*(p,i),(p,i+1)*> | ∀*p*, ∀*i*: 0 ≤ *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* → {red, blue} by

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

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* ≤ *j* ≤ *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, $(r,t) \notin G(p)$, 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 $|\{r | r \neq p\}| = n - 1$.

**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, $e$ 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 $M$, $|M' - |M| \geq |M| - 1$. 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 blocking and nonblocking protocols. While blocking protocols achieving the lower bound are well-known, it had been previously conjectured that this bound was too
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 \(p_i\), then \(p_i\) 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 \(p_i\) receives no requests for help before step \(2j + 2\); thus, \(p_i\) 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. \(\square\)

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 \(2i + 2\). \(\square\)

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. \(\square\)

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. \(\square\)

This completes the proof of theorem 2. \(\square\)

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
accounted, 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 commit-
table at step $a$ if and only if for all processors $q$, $(p,r) \in G(q)$, i.e., if and only if $a$ has been reached by all processors.

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

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

**Justification:** Let $z, \not\in \in M$. 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, $z$ can know at $r$ that the message was sent only if $z$ sent it. Since $z$ can send to at most one processor during step $r-1$, there is at most one processor (other than $z$) in $M$ known
by $z$ 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 $r, \ldots, r+k$ 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 of 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,

$$(z,i) \in H(p) \iff \begin{cases} \text{((z,i-1),(p,i))} \in G & \text{or} \quad \text{if } i \geq r \\ \text{(z,i)} \in D(v) \text{ for some } v \in H(p) & \text{otherwise}. \end{cases}$$

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,

$$(z,i) \in F(p) \iff \begin{cases} p \text{ sends to } z \text{ at } i-1 \text{ in } J & \text{or} \quad \text{if } i > r \\ \text{(z,i)} \in D(v) \text{ for some } v \in F(p) & \text{otherwise}. \end{cases}$$

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,

$FGH(p) = F(p) \cup G(p) \cup H(p)$.

**Specification of J:** For all $x \not\in p$, $x$ fails just prior to step $\mu i \{ (z,i) \in FGH(p) \}$, denoted $fail(x)$. 

Claim: If \((x,i) \not\in FGH(p)\) 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 \(z, (z,t) \in FGH(p)\) in run \(J\).

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

Let \(\alpha = \max \{f(\text{fail}(x))\} - r\) 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 processes 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,

\[
f(i) = 2f(i-1) + 1 = 2(2^{i-1}(d+2) - 1) = 2^i(d+2) - 1.
\]

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

\[
2^\alpha(d+2) - 1 \geq n
\]

\[
\Rightarrow 2^\alpha(d+2) > n
\]

\[
\Rightarrow 2^\alpha > n/(c(k+1)+2)
\]

\[
\Rightarrow 2^\alpha > (n/c)/(k+1+2/c)
\]

\[
\Rightarrow \alpha > \log(|M| - \log(k+1+2/c)).
\]

But \(c > 1\) and \(|M| \geq n/t\), so

\[\alpha \geq \log(n/t) - \log(k + O(1)),\]

whence,

\[1 \leq \log(n/t) - \loglog n - O(1),\]

since \(k < t \leq 2\log n\).

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

\[2 \log(n/t) + \alpha \leq r + \alpha \leq r + k = t.
\]

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

\[2\log n - \loglog n - O(1) \leq t.
\]

\[\square
\]

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 availability 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 commitable state, whose existence is dictated by the nonblocking rules N1 and N2. For Protocol A, the length of the commitable state is exactly nΔ.

Protocol A exhibits an important property: once the reporting phase begins, a commitable transaction is committed so long as a single processor in the commitable state remains operational. This can be restated as a local condition for each processor: once commitable, the processor will eventually commit the transaction or fail. Hence, for each processor p, the predicate commitable(p) ∨ 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 commitability. 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 commitable (but not committed) results and then delays voting until commitment of the read results is certain (nΔ in Protocol A). The situation is depicted in Figure 2b.

The ordering of events ensures:

1. transition of the first subtransaction into the commitable 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 commitable state, the first aborts only if the host processor fails, and in this case, the second subtransaction would independently and explicitly 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.

<table>
<thead>
<tr>
<th>initial (distribution)</th>
<th>voted (precommit)</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. A blocking subtransaction.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initial (distribution)</th>
<th>voted (precommit)</th>
<th>committable</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>b. A nonblocking subtransaction.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.)
The major network requirement for the extension is that processor failures be detected and failure information be propagated in a timely fashion. Specifically, the elapsed time between the failure of processor $i$ and the receipt of the failure notification by 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 additional $F$ time units, where $F$ is the maximum delay for the second host to learn of the failure of the first. For sizable subtransactions, the initial phase is computationally intensive and may well exceed the required delay time.

6. Discussion

We have shown that there is no fundamental difference in message complexity between blocking and nonblocking protocols. This surprising result has relegated a contrary "folk theorem" to its proper place in mythology. Moreover, the proof is in the form of interesting new nonblocking protocol.

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

7. References


APPENDIX

B. Protocol A.

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

\[
\text{f1}
\]

\[
\text{if } (\text{step} = 0) \rightarrow
\]

\[
\text{if } \text{vote}(i) = \text{abort} \rightarrow \{\text{unilateral abort}\}
\]

\[
\text{decided}(i), \text{decision}(i) := \text{true}, \text{abort};
\]

\[
\text{vote}(i) = \text{commit} \rightarrow \text{decision}(i) := \text{false};
\]

\[
\text{f1}
\]

Send vote$(i)$ to $p_0$;

\[
\text{f1}
\]

\[
1 < \text{step} \leq i \rightarrow \text{skip};
\]

\[
i < \text{step} \leq n + i \rightarrow
\]

\[
\text{f1} \quad (\sim \text{decided}(i)) \rightarrow
\]

\[
\text{f1}
\]

\[
\text{if } (\text{decision message received}) \rightarrow
\]

\[
\text{decided}(i) := \text{true};
\]

\[
\text{decision}(i) := \text{message};
\]

\[
\text{if } (\text{no decision message received}) \rightarrow
\]

\[
\text{if } (j < i) \rightarrow
\]

\[
\text{write "help" to } p_j;
\]

\[
\text{if } (j = i) \rightarrow \text{skip};
\]

\[
\text{if } (j = i + 1) \rightarrow
\]

\[
\text{decided}(i) := \text{true};
\]

\[
\text{decision}(i) := \text{abort};
\]

\[
\text{f1}
\]

\[
\text{f1}
\]

\[
(\text{decided}(i) \land \text{"help" message received}) \rightarrow
\]

\[
\{\text{message is from } p_{\text{dec} - i - 1}\}
\]

\[
\text{send decision}(i) \text{ to } p_{\text{dec} - i - 1}
\]

\[
(\text{decided}(i) \land \text{no "help" message received}) \rightarrow
\]

\[
\{p_{\text{dec} - i - 1} \text{ failed or decided}\} \text{ skip;}
\]

\[
\text{f1}
\]

\[
\text{step} = n + i + 1 \rightarrow \text{act according to decision}(i)
\]

\[
\text{step} > n + i + 1 \rightarrow \text{skip}
\]

\[
\text{f1}
\]

The algorithm for $p_1$ is similar, but if $p_1$ does not hear from $p_0$ during step 2 it knows the result of the vote.
has been lost, and can decide to abort at that time.

The algorithm for $p_0$ is simple:

\[
\text{if } \text{step} = 0 \rightarrow \text{skip;}
\]

\[
\text{if } \text{step} = 1 \rightarrow
\]

\[
\text{if } (n-1 \text{ commit votes are received})
\]

\[
\land \text{vote}(0) = \text{commit} \rightarrow
\]

\[
\text{decision}(0) := \text{commit;}
\]

\[
\text{if } (\text{fewer than } n-1 \text{ commit votes received})
\]

\[
\lor \text{vote}(0) = \text{abort} \rightarrow
\]

\[
\text{decision}(0) := \text{abort;}
\]

\[
\text{if } 1 \leq \text{step} < n \rightarrow \text{write decision}(0) \text{ to } p_{\text{dep;}}
\]

\[
\text{if } \text{step} \geq n \rightarrow \text{skip;}
\]