Bounds on Fundamental Problems in Parallel and Distributed Computation

Cynthia Dwork
Ph.D. Thesis

TR 84-582
January 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

now at:
Laboratory for Computer Science
MIT NE43-520a
545 Technology Square
Cambridge, Massachusetts 01239

Bounds on Fundamental Problems in Parallel and Distributed Computation
(Revised 1/84)

Cynthia Dwork

Work performed at Cornell University.

Acknowledgements

John Hopcroft and David Gries have provided extraordinary support during hard times. Without them, this thesis could never have been begun.

I am indebted to John Hopcroft for many helpful discussions. Thanks also go to John Gilbert, Nancy Lynch, Dale Skeen, and Larry Stockmeyer for their interest and criticisms.

Contents

List of Figures

CHAPTER 1

Introduction

The principal motivation for this work is an interest in lower bounds on communication for parallel computation. The search for lower bounds has invariably yielded interesting, new upper bounds.

In chapter 2, we show that a parallel RAM with no write conflicts allowed requires $\Omega(\log n)$ steps to compute any n-ary function that depends on all its inputs, such as the logical OR of n bits. The result is of interest because it is possible to transmit information by not writing, and this observation leads to an algorithm that beats the intuitive lower bound of $\log_2 n$ steps.

The use of nonwrites to transmit information is not restricted to PRAMs with completely reliable processors, but extends to certain distributed synchronous models in which processors may be faulty. Using nonwrites to replace writes yields improved nonblocking protocols for transaction commitment. In this problem, the processors in the network are notified of a possible database transaction. Each processor votes either to commit or to abort the transaction. The votes are tallied and the result is distributed to the participants, which act accordingly. The difficulty is to ensure consistency even if some of the processors are faulty: there can be no two processors, one of which commits the transaction while the other aborts. A protocol is *nonblocking* if an operational processor never suspends execution ("blocks") pending the recovery of a failed processor. While it was known that $2(n-1)$ messages suffice for a blocking protocol, the lower bound for nonblocking protocols was thought to be about 50% higher. This conjecture was disproved by modifying an algorithm in [26] by replacing certain messages with nonwrites. A nonblocking commitment protocol with best-case message complexity $2(n-1)$ was obtained. The algorithm appears in chapter 4, together with a proof of its optimality with respect to message complexity for both nonblocking and blocking protocols. Chapter 4 also contains results on time: we prove that nonblocking protocols are inherently more time consuming than blocking protocols.

Finally, the chapter looks at the problems of reliable and atomic broadcast. By definition any protocol for atomic broadcast solves the reliable broadcast problem. We prove computational equivalence of the two problems, arguing that any protocol for reliable broadcast can be trivially transformed into a protocol for atomic broadcast. The transformation affects neither the number of messages sent nor the number of steps taken by any processor. This disproves yet another popular conjecture: that atomic broadcast is more costly to achieve than reliable broadcast.

Some of the groundwork for chapter 4 appears in chapter 3, which deals with a related problem from a completely different angle. Fischer, Lynch and Paterson ([13]) showed that consensus cannot be achieved in any sufficiently asynchronous model of distributed computation if even one processor can be faulty. The assumptions of their model are varied in [6] in an effort to understand the precise reason for impossibility. Five critical system parameters are identified in that paper, and maximum resiliency for all resulting 32 cases has been found.

The boundaries between existence of consensus algorithms and impossibility are very sharp: either there is no t-resilient protocol, for $t \geq 2$, or there is an n-resilient protocol, where n is the number of processors in the system. We therefore chose one model in which no protocol exists and asked how little must be added in order to obtain k-resiliency, for arbitrary k, $1 \leq k \leq n$. Another sharp boundary was found. These results are summarized in chapter 3, wherein some algorithms and impossibility results are presented.

Chapter 5 deals with matrix transposition networks. These are networks with n input vertices and n output vertices. Vertices correspond to *registers*. Associated with each input register are n bits, viewed as a row of an $n \times n$ Boolean matrix. The problem is to transpose the matrix, so that each output register holds a row of the transposition. The problem is made nontrivial by imposing a maximum capacity of n bits on each register. An edge/depth tradeoff of $\theta(kn^{1+1/k})$ edges for oblivious, conservative networks of depth k is proved. An interesting open question is whether the lower bounds apply to nonconservative networks, that is, networks in which bits may lose their identity. We show the lower bound holds for nonconservative networks of depth at most 3.

CHAPTER 2

Time Bounds for Computation of Simple Functions on a PRAM

1. Introduction

In this chapter we prove that a parallel random access machine (PRAM) in which no write conflicts are allowed requires $\Omega(\log n)$ steps to compute the Boolean OR of n bits stored in the first n cells of shared memory. The bound is independent of the number and computational power of the individual processors. The result is interesting because the intuitive lower bound of $\log_2 n$ can be beaten by a constant multiplicative factor.

A *PRAM* (adapted from [14]) is a collection of independent, arbitrarily powerful processors that communicate through a common memory M. Any number of processors may simultaneously read from a given cell of M, but simultaneous write access is forbidden: if two processors simultaneously try to write into the same common memory location, the system fails. Access to shared memory is synchronized in a lockstep fashion. A *computation* of a PRAM is the parallel execution by every processor in the system of

```
do <condition> →
     Read from at most 1 cell of M;
     Compute (privately);
     Write into at most 1 cell of M
od
```

Each iteration of the loop is called a *step*. In the case of logical OR, or any other n-ary function, the first n cells of shared memory, $M[1],\ldots,M[n]$, are initialized with the values of the inputs $x_1,\ldots,x_n$, respectively. The PRAM is said to *run in time T* if each processor executes at most T steps.

2. Upper Bounds

Let us first consider the naïve approach to computing OR. Let each processor p_i, $1 \leq i \leq 2^v = n$, have a private variable y_i, initially set to zero. At step 0, let p_i read $M[i]$ and set y_i to the value read. At step t, $1 \leq t \leq v$, processor p_i executes:

if $i \equiv 1 \pmod{2^t}$ **then**

$\{y_i = M[i] = x_i \vee \ldots \vee x_{i+2^{t-1}-1}\}$

$\{M[i+2^{t-1}] = x_{i+2^{t-1}} \vee \ldots \vee x_{i+2^t-1}\}$

$y_i := y_i \vee M[i+2^{t-1}]$;

$\{y_i = x_i \vee \ldots \vee x_{i+2^t-1}\}$

$M[i], t := y_i, t+1$

$\{y_i = M[i] = x_i \vee \ldots \vee x_{i+2^{t-1}-1}\}$

fi

The number of bits "known" to p_i and $M[i]$ doubles at each step, so this simplistic approach leads to an n-processor algorithm requiring $1 + \log_2 n$ steps to compute the OR of n bits. However, it is possible to do better than this. Let F_j be the jth Fibonacci number, i.e., $F_0 = 0$, $F_1 = 1$, and, for $j > 1$, $F_j = F_{j-1} + F_{j-2}$. We can compute the OR of F_{2T+1} inputs using F_{2T+1} processors in time T.

Consider the general situation shown in figure 2.1, temporarily ignoring the particular choice of w, j, and k relative to i. We have

$$y_i = x_i \vee \ldots \vee x_{j-1}$$

$$M[j] = x_j \vee \ldots \vee x_{k-1}$$

$$M[w] = x_w \vee \ldots \vee x_{i-1}$$

Let p_i compute $y_i := y_i \vee M[j]$. If $y_i = 1$, then $M[w] \vee y_i = 1$, since for any x, $x \vee 1 = 1$. In this case, regardless of whether or not p_i knows the contents of $M[w]$, if it writes a 1 into $M[w]$, then $M[w]$ will the contain $x_w \vee \ldots \vee x_{k-1}$. However, if $y_i = 0$, then $M[w] \vee y_i = M[w]$ already, since for any x, $x \vee 0 = x$. This observation suggests the algorithm of theorem 2.1.

Theorem 2.1: The logical OR of F_{2T+1} inputs can be computed in time T using F_{2T+1} processors.

Proof: Algorithm FAST_OR, which appears in figure 2.2, achieves these bounds. A typical step is shown in figure 2.1. Correctness and running time follow immediately from the annotations, which reflect the assumption of a completely synchronous system of reliable processors. Since p_1 takes no steps $F_{2T+1} - 1$ processors suffice.

Remark: Since $F_{2t} \sim [(\frac{1+\sqrt{5}}{2})^2]^t > 2^t$, the algorithm beats the intuitive lower bound of $\log_2 n$ steps.

---

w	i	j	k
$M[w] = \vee\ (r : w \leq r < i : x_r)$		$M[j] = \vee\ (r : j \leq r < k : x_r)$	

At step t p_i executes:
$\{M[w] = \vee\ (r : w \leq r < i : x_r), M[j] = \vee\ (r : j \leq r < k : x_r)\}$
$\{y_i = \vee\ (r : i \leq r < j : x_r)\}$
$y_i := y_i \vee M[j];$
if y_i **then** $M[w] := y_i$ **fi**
$\{M[w] = \vee\ (r : w \leq r < k : x_r),\ y_i = \vee\ (r : i \leq r < k : x_r)\}$

$$w + F_{2t+1} = i\ ;\ i + F_{2t} = j\ ;\ j + F_{2t+1} = k$$

Figure 2.1: Typical Step of Algorithm FAST_OR

{Protocol for p_i : $2 \leq i \leq F_{2T+1}$}

{$n = F_{2T+1}$}

{$X(i,j) \equiv x_i \vee \ldots \vee x_j$}

{$r(i,t) \equiv \min(n, i+F_{2t}-1)$}

{$s(i,t) \equiv \min(n, i+F_{2t+1}-1)$}

$t, y := 0,0;$

do $t < T \rightarrow$

  {$y_i = X(i,r(i,t))$}

  if $i+F_{2t} \leq n \rightarrow$

    {$M[i+F_{2t}] = X(i+F_{2t}, s(i+F_{2t},t))$}

    $y_i := y_i \vee M[i+F_{2t}]$

    {$y_i = X(i,s(i+F_{2t},t))$}

  $[]$ $i+F_{2t} > n \rightarrow$ {$y_i = X(i,r(i,t)) = X(i,n) = X(i,s(i+F_{2t},t))$}

    skip

  fi

  {$y_i = X(i,s(i+F_{2t},t))$}

  if $i > F_{2t+1}$ **then**

    {$M[i-F_{2t+1}] = X(i-F_{2t+1}, s(i-F_{2t+1},t))$}

    if y_i **then** $M[i-F_{2t+1}] := 1$ **fi**

    {$M[i-F_{2t+1}] = X(i-F_{2t+1}, s(i+F_{2t},t))$}

  fi

  {$i+F_{2t+1} \leq n \Rightarrow M[i] = X(i,s(i+F_{2t+2},t)) = X(i,s(i,t+1))$}

  {$i+F_{2t+1} > n \Rightarrow M[i] = X(i,s(i,t)) = X(i,n) = X(i,s(i,t+1))$}

  {$y_i = X(i,s(i+F_{2t},t)) = X(i,r(i,t+1))$}

  $t := t+1$

  {$M[i] = X(i,s(i,t))$}

  {$y_i = X(i,r(i,t))$}

od

Figure 2.2: Algorithm FAST_OR

Corollary 2.1: The logical AND of F_{2T+1} inputs can be computed in time T using F_{2T+1} processors.

Proof: The algorithm, FAST-AND, is a trivial modification of algorithm FAST-OR.

Corollary 2.2: Let $n = F_{2T+1}$ and let $f : \{0,1\}^n \to \{0,1\}$ be any n-ary Boolean function with k satisfying assignments to its arguments. Then f can be computed in time $T+2$ using kn processors and kn cells of common memory.

Proof:

Let $A_1,...,A_k$ denote all the assignments to the inputs $x_1,...,x_n$ for which f has value 1. With each $A_i = <a_{i1}, \ldots, a_{in}>$ associate a set $S_i = \{p_{i1},...,p_{in}\}$ of processors and a set $C_i = \{c_{i1},...,c_{in}\}$ of cells in M satisfying

$$\mathbf{A}\ (i,j : 1 \leq i,j \leq k : S_i \bigcap S_j = C_i \bigcap C_j = \emptyset)$$
$$\mathbf{A}\ (j : 1 \leq j \leq n : c_{1j} \text{ is identically } M[j]).$$

Let I be an assignment to the inputs of f. To compute $f(I)$ each processor p_{ij} begins by executing

$y_{ij} := M[j];$

if $y_{ij} = a_{ij} \to c_{ij} := 1$

$[]$ $y_{ij} \neq a_{ij} \to c_{ij} := 0$

fi

If $\mathbf{E}\ (i : 1 \leq i \leq k : I = A_i)$, then i is unique and C_i contains n ones. In the next T steps each set S_i computes $\Lambda(j : 1 \leq j \leq n : c_{ij})$ in the partition C_i, using algorithm FAST-AND. Note that after the first $T + 1$ steps $M[1] = 1$ if and only if $I = A_1$, and in general $c_{i1} = 1$ iff $I = A_i$, $1 \leq i \leq k$. Finally, each p_{i1} executes

$y_{i1} := c_{i1}$;

if $y_{ij} \rightarrow M[1] := 1$

$[]$ $\overline{y_{ij}} \rightarrow$ skip

fi

If I is an input configuration for which f has value 1, then after $T + 2$ steps $M[1] = 1$, otherwise, $M[1] = 0$.

3. Lower Bounds

In addition to exploiting the properties of the logical OR the improved algorithm raises a crucial point: *in any sufficiently synchronous system information can be transmitted by a non-write*. Now, for a fixed cell c of M, many processors can simultaneously "not-write" to c. The problem is to bound the amount of information transmitted by several simultaneous "non-writes" to the same location. The bound obtained below is independent of the size of shared memory and the size of a cell, as well as of the number of processors and their individual computational prowess.

In order to discuss non-writes in a precise way we further specify the definition of a PRAM.

Definition: A *PRAM* consists of a set $\Pi = \{p_1, p_2, \ldots\}$ of processors, a shared memory M with cells $M[1], M[2], \ldots$, a possibly infinite alphabet Σ, a number n of inputs, and an execution time T. Each processor p_i consists of a state set Q_i and functions $\rho_i\colon Q_i \rightarrow N$, $\tau_i\colon Q_i \rightarrow N$, $\sigma_i\colon Q_i \rightarrow \Sigma$, and $\delta_i\colon Q_i \times \Sigma \rightarrow Q_i$. For $q \in Q_i$, $\rho_i(q)$ is the index of the next cell to be read, $\tau_i(q)$ is the index of the next cell to be written into ($\rho_i(q) = 0$ and $\tau_i(q) = 0$ indicate no cell is read from or written into, respectively), $\sigma_i(q)$ is the symbol written, and δ_i is the state transition function.

For any $t \leq T$, let q_i^t and s_i^t denote the state of processor p_i and the contents of $M[i]$, respectively, at time t. Thus on inputs $\{x_1, \ldots, x_n\}$ $s_i^0 = x_i$ for all $1 \leq i \leq n$, while

$s_i^0 = *$ for all $i > n$, where $*$ is a distinguished element of Σ. Let $q_i^0 = q_0$ for some distinguished start state $q_0 \in Q_i$.

The state of a processor p_i at the beginning of step $t+1$ is a function of its state at the beginning of step t and the contents of the cell read during t. Thus,

$$q_i^{t+1} = \delta_i(q_i^t, s_j^t), \text{ where } j = \rho_i(q_i^t).$$

The write operation is destructive:

$$s_i^{t+1} = \begin{cases} \sigma_j(q_j^{t+1}) & \text{if } \tau_j(q_j^{t+1}) = i \\ s_i^t & \text{if } \mathbf{A}(j :: \tau_j(q_j^{t+1}) \neq i) \end{cases}$$

It is a condition of correctness of the PRAM that at each step, at most 1 processor writes into cell $M[i]$, so

$$\mathbf{A}(t,j,k : j \neq k : \tau_j(q_j^{t+1}) \neq \tau_k(q_k^{t+1}) \ \vee \ \tau_j(q_j^{t+1}) = 0).$$

Finally, $s_1^T = f(x_1,...,x_n)$, where f is the function computed by the PRAM.

Let $I = \langle a_1,...,a_j,...,a_n \rangle$ denote an assignment to the inputs of a PRAM. Then $s_c^t(I)$ and $q_i^t(I)$ denote s_c^t and q_i^t, respectively, in instance I, and $I(j,b) = \langle a_1,...,a_{j-1},b,a_{j+1},...,a_n \rangle$ denotes the assignment agreeing with I on all but the j^{th} input, which has value b.

Definition: An input x_j is said to *affect* p *at* t *with* I if and only if for some $b \in \Sigma$ $q_i^t(I) \neq q_i^t(I(j,b))$. That is, x_j affects p at t with I whenever there exists an assignment b to x_j such that the state of p_i at time t with initial assignment I differs from its state at time t with assignment $I(j,b)$. This is the metric for *information known* to p at t with I. An analogous definition applies to cells.

Let $L(p,t,I)$ (respectively, $K(c,t,I)$) be the set of indices of inputs affecting processor p (respectively, cell c) at time t with I.

Let f be an n-ary function. For each input $D = \langle d_1,...,d_n \rangle$ let $S_{f,D}$ denote the set of "critical indices" for f with respect to D:

$$S_{f,D} = \{i: 1 \leq i \leq n : \mathbf{E}(b : b \in \Sigma: f(D) \neq f(D(i,b)))\}.$$

Then for any PRAM P computing f with input D in time T we have

$$S_{f,D} \subseteq K(M[1],T,D).$$

In particular, if f is the logical OR of n Boolean variables then

$$\{1,...,n\} \subseteq K(M[1],T,<0,...,0>).$$

Theorem 2.2: Let f be an n-ary function, let $k = \max(D : D \in \Sigma^n : |S_{f,D}|)$. Then any PRAM computing f on input D requires at least $\log_a k$ iterations, where $a = \frac{5+\sqrt{21}}{2}$.

Proof: Let P be a PRAM computing f. By choice of f there exists an input configuration I such that $|S_{f,I}| = k$. Thus all k inputs x_j such that j is in $S_{f,I}$ must eventually affect the "output" cell $M[1]$ of P on input configuration I. The theorem is proved by bounding the number of inputs affecting any cell at time t with I, for all times t. This bound is given in terms of the following recursive definitions of functions $P(t)$ and $C(t)$:

Definition:

$$\begin{aligned}
P(0) &= 0 \\
C(0) &= 1 \\
P(t+1) &= P(t) + C(t) \\
C(t+1) &= 3P(t) + 4C(t).
\end{aligned}$$

In lemma 2.1 we prove that C can actually be defined as

$$C(t+1) = \frac{3+\sqrt{21}}{2\sqrt{21}}\left(\frac{5+\sqrt{21}}{2}\right)^t + \frac{\sqrt{21}-3}{2\sqrt{21}}\left(\frac{5-\sqrt{21}}{2}\right)^t.$$

In lemma 2.2 we prove the following lower bounds on P and C in terms of L and K:

$$P(t) \geq \max(p,I :: |L(p,t,I)|)$$

$$C(t) \geq \max(c,I :: |K(c,t,I)|).$$

So by lemmas 2.1 and 2.2 and the fact that $S_{f,I}$ is contained in $K(M[1],T,I)$,

$$|S_{f,I}| \leq |K(M[1],T,I)| \leq C(T) \leq a^T .$$

Thus, $T \geq \log_a |S_{f,I}| = \log_a k$.

Lemma 2.1:

$$C(t+1) = \frac{3+\sqrt{21}}{2\sqrt{21}}\left(\frac{5+\sqrt{21}}{2}\right)^t + \frac{\sqrt{21}-2}{2\sqrt{21}}\left(\frac{5-\sqrt{21}}{2}\right)^t.$$

Proof: $P(t+1)$ and $C(t+1)$ may be expressed in matrix form as

$$\begin{pmatrix} P(t+1) \\ C(t+1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} P(t) \\ C(t) \end{pmatrix}.$$

Thus,

$$P(t+1) = \alpha a^t + \beta b^t$$
$$C(t+1) = \gamma a^t + \delta b^t$$

where $a = \dfrac{5+\sqrt{21}}{2}$ and $b = \dfrac{5-\sqrt{21}}{2}$ are the eigenvalues of $\begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix}$ and the constants $\alpha = -\beta = \dfrac{1}{\sqrt{21}}$ and $\gamma = 1-\delta = \dfrac{3+\sqrt{21}}{2\sqrt{21}}$ are determined by the initial conditions.

Lemma 2.2:

$$P(t) \geq \max\ (p,I :: |L(p,t,I)|)$$

$$C(t) \geq \max\ (c,I :: |K(c,t,I)|).$$

Proof:

The case $t = 0$ is immediate, since processors initially know nothing and each cell can be affected by at most one input. Thus,

$$|L(p,0,I)| = 0 = P(0)$$
$$|K(c,0,I)| \leq 1 = C(0).$$

Assume the result for t inductively. At each step a processor can read from at most one cell, so by induction and the definition of $P(t+1)$ we have

$$|L(p,t+1,I)| \le |L(p,t,I)| + |K(c,t,I)| \le P(t) + C(t) = P(t+1).$$

The inductive step for a cell c is more complicated. There are two cases to consider, according to whether or not c is written into during step t with I. If p writes into c at t, then $K(c,t+1,I) = L(p,t+1,I)$, since the write is destructive, so in this case

$$|K(c,t+1,I)| = |L(p,t+1,I)| \le P(t+1) \le C(t+1).$$

In the case of no write, not only is the information in c at the beginning of step t preserved at $t+1$, but the information that no processor writes to c at t is added. Let I, c, and t be fixed. Then

$$|K(c,t+1,I)| = |K(c,t,I)| + |K(c,t+1,I) - K(c,t,I)|.$$

If no processor writes into c at t with I and j is in $K(c,t+1,I)$ but is not in $K(c,t,I)$, then by definition of K there exists a value b satisfying

$$s_c^t(I(j,b)) = s_c^t(I) = s_c^{t+1}(I) \ne s_c^{t+1}(I(j,b))$$
$$=> s_c^t(I(j,b)) \ne s_c^{t+1}(I(j,b)).$$

Thus some processor writes to c at t with $I(j,b)$. In lemma 2.3 we show there can be at most $3P(t+1)$ such indices j. Thus,

$$\begin{aligned}|K(c,t+1,I)| &\le |K(c,t,I)| + 3P(t+1)\\ &\le C(t) + 3P(t+1)\\ &= 3P(t) + 4C(t)\\ &= C(t+1).\end{aligned}$$

This completes the proof of lemma 2.2.

Lemma 2.3: Let $U = \{u_1,\dots,u_r\}$ be the set of distinct indices for which there exist (not necessarily distinct) processors $z_1,\dots,z_r$ such that

$$\mathbf{E}\ (b_i : b_i \in SIGMA:\ z_i \text{ writes into } c \text{ at } t \text{ with } I(u_i,b_i)).$$

Then $|U| = r \leq 3P(t+1)$.

Proof:

Let $B = (U,V,E)$ be a directed bipartite graph with vertex sets $U = \{u_1,...,u_r\}$ and $V = \{v_1,...,v_r\}$, where the v's are any r distinct new objects. The edge (u_k,v_j) is in E if and only if for some b_j u_k affects z_j with $I(u_j,b_j)$ at $t+1$ (i.e., at the write stage of step t).

Each processor z_i is affected by at most $P(t+1)$ inputs at the write stage of step t, so the indegree of v_i is bounded above by $P(t+1)$, and $|E| \leq r\,P(t+1)$. On the other hand, correctness of P ensures that if z_k and z_j are distinct processors, then u_k affects z_j with $I(u_j,b_j)$ at t, or u_j affects z_k with $I(u_k,b_k)$ at t. In other words, z_j can distinguish $I(u_j,b_j)(u_k,b_k)$ from $I(u_j,b_j)$ or z_k can distinguish $I(u_j,b_j)(u_k,b_k)$ from $I(u_k,b_k)$. Otherwise, both processors would write to c at t on input configuration $I(u_j,b_j)(u_k,b_k)$. Thus if $z_k \neq z_j$, at least one of (u_k,v_j) and (u_j,v_k) is in E.

There are r choices for u_j. For each of these, there are at most $P(t+1)$ choices of u_k such that $z_j = z_k$, so there are at least $r - P(t+1)$ choices for u_k such that $z_j \neq z_k$. We therefore have

$$\frac{r(r-P(t+1))}{2} \leq |E| \leq r\,P(t+1),$$

$$\Rightarrow r - P(t+1) \leq 2P(t+1)$$

$$\Rightarrow r \leq 3P(t+1).$$

Corollary 2.3: The following functions require at least $\log_a n$ time on a PRAM, where $a = \frac{5+\sqrt{21}}{2}$: (1) The logical OR of n bits; (2) the logical AND of n bits, (3) the maximum of n integers; (4) undirected graph accessibility: $UGAP(A,s,t) = 1$ if and only if there is a path from vertex s to vertex t in the undirected graph with adjacency matrix A.

Remarks:

Most of these results appear in [3], [4], and [23]. In [3], the proof of the lower bound was contributed by Steve Cook. The central observation leading to corollary 2.1 was made by John Hopcroft. [4] contains additional related results.

CHAPTER 3

Distributed Consensus

1. Introduction (adapted from [6])

The ability of separated processors to reach agreement is a fundamental problem of both theoretical and practical importance in the area of distributed computing ([9], [10a], [18], [20]). We consider a system of N processors $p_1,...,p_N$, which communicate by sending messages to one another. Initially, each p_i has a binary value x_i. At some point during its computation, a processor can irreversibly decide on a binary value v. Each processor follows a deterministic protocol involving the receipt and sending of messages. Even though the individual processors behave deterministically, there are three potential sources of indeterminism in the system. Processors might run at varying speeds, it might take varying amounts of time for messages to be delivered, and messages might be received in an order different from the one in which they were sent.

A protocol solves the *weak consensus* problem if (i) no matter how the system runs, every nonfaulty processor makes a decision after a finite number of steps, (ii) no matter how the system runs, two different processors never decide on different values, and (iii) 0 and 1 are both possible decision values for (possibly different) assignments of initial values.

If the processors and the communication system are completely reliable, the existence of consensus protocols is trivial. The problem becomes interesting when the protocol must operate correctly even if some processors are faulty. So far, we have only investigated benign failures, in which faulty processors take no steps and neither send nor receive messages. A consensus protocol is *t-resilient* if it operates correctly when at most t processors fail.

Our point of departure is the interesting result of Fischer, Lynch, and Paterson ([13]) which states that in a completely asynchronous system no consensus protocol is 1-resilient; that is, even one failure cannot be tolerated. In reading their proof one sees that three different types of asynchrony are used:

Processor asynchrony allows processors to "go to sleep" for arbitrarily long finite amounts of time while other processors continue to operate;

Communication asynchrony precludes an *á priori* bound on message delivery time;

Message order asynchrony allows messages to be delivered in an order different from the one in which they were sent.

One goal of this work was to determine how much asynchrony is needed to obtain impossibility. Specifically, is it necessary for all three types of asynchrony to be present simultaneously? We find it is not. In fact, using the definition in [13] of an *atomic step*, in which a processor can attempt to receive a message and depending on the value received, if any, it can change its internal state and send (possibly distinct) messages to all the other processors, we prove either synchronous communication or synchronous message order alone to be sufficient for the existence of an n-resilient consensus protocol. (If all n processors fail, then the nonfaulty processors reach agreement trivially.) By contrast, lock-step processor synchrony is insufficient: we prove the absence of even a 1-resilient protocol in this model, thus strengthening the main result of [13]. Our n-resilient protocols are fairly delicate and depend on the definition of an atomic step of a processor. For example, in the case where we have synchronous communication, if receiving and sending are split into two separate operations so that a failure can occur in between, then the n-resilient protocol fails, and in fact [6] proves that there is no 1-resilient protocol in this case. Similarly, if a processor can send a message to at most one other processor in an atomic step (we call this point-to-point transmission) then there is a 1-resilient protocol but no 2-resilient protocol.

Five critical parameters are identified in [6] (only the first four are discussed here):

(1) processors: synchronous or asynchronous,

(2) communication: synchronous or asynchronous,

(3) message order: synchronous or asynchronous,

(4) transmission: broadcast or point-to-point, and

(5) receive/send: atomic or separated.

Varying these yields 32 cases, and [6] finds the maximum resiliency for each case. More interestingly, there are four cases in which n-resilient protocols exist, but any weakening of the system by changing one parameter destroys all possibility of even a t-resilient protocol, where t is either 1 or 2. Thus the boundary between possibility and impossibility of solving the consensus problem is very sharp. We find another type of boundary by allowing a processor to broadcast to k processors in an atomic step. This *k-casting* is said to be *serializable* if whenever processors p and q k-cast messages $m1$ and $m2$, respectively, to processors r and s, then r and s receive the two messages in the same order. In a system with asynchronous processors and asynchronous communication, we show, for any $1 \leq k \leq n$, that serializable k-casting is sufficient for $(k-1)$-resiliency. We also show it is necessary, in the sense that serializable $(k-1)$-casting and unserializable k-casting are both insufficient for $(k-1)$-resiliency.

Our principle motivation was to understand intuitively why Fischer, Lynch and Paterson were able to prove impossibility, and to develop heuristic arguments to allow one to make an educated guess of the maximum resiliency before actually proving it. The basic intuition is that if letting t processors fail can effectively "hide" an event or the relative ordering among several events, then no consensus protocol is t-resilient. In the original proof in [13], the event is a "critical step" where one processor p takes a step which moves the system from some configuration C_0 to some configuration C_1, and such that if the other processors must reach a decision starting from C_i then i is the only possible decision value, $i = 0,1$. Whether or not p fails, the effect can be temporarily hidden from the other processors, since even if p takes the critical step, the communication system can temporarily withhold all the messages sent by p during the critical step. However, if we have a fixed upper bound on message delivery time, or if message order cannot be scrambled, then these messages cannot be hidden indefinately; this explains intuitively why we get n-resiliency in these two cases. The heuristic argument also explains why we get 1-resiliency but not 2-resiliency in the case of bounded message delivery time and point-to-point transmission. In the critical step, p sends a message to a single processor q. To hide this event, it is necessary and sufficient that both p and q fail.

Finally, we note the existence of various probabilistic results. If the model in [13] is modified only to ensure fairness, Bracha and Toueg ([2]) describe a consensus protocol that tolerates up to $\lfloor\frac{n-1}{2}\rfloor$ benign failures, and they prove the impossibility if a majority of the processors are faulty. They prove a similarly tight result for up to $\lfloor\frac{n-1}{3}\rfloor$ malicious failures. The algorithms of [2] closely resemble related work of Ben-Or ([1]). For Byzantine failures Ben-Or only obtains $\lfloor\frac{n-1}{5}\rfloor$-resiliency, but he allows for a particularly nasty message system, as well as malicious process failures. Related results have also been obtained by Rabin ([22]).

2. System Parameters (adapted from [6])

In defining the system parameters it is useful to imagine that one is standing outside the system holding a "real time clock" that ticks at a constant rate. At each tick of the real time clock, at most one processor can take a step. The processors are modeled as infinite-state machines. In the most general definition of *step*, a processor can attempt to receive a message, and based on the value of the received message (or based on the fact that no message was received) it can change its state and broadcast a message to all processors. Restrictions on this definition appear below. The letters U and F below refer to situations that are unfavorable or favorable, respectively, for solving the consensus problem. The letters in parentheses refer to the variable name associated with each parameter. The results of [6] are summarized in the

Karnaugh-like map in figure 3.1.

Processors (p)

U *Asynchronous:* Any processor can wait arbitrarily but finitely many real time steps between its own steps (i.e., there is no bound on the rate of drift of the internal clocks of the processors). If a processor takes only finitely many steps in an infinite run of the system, then it has failed.

F *Synchronous:* There is a constant $\Phi \geq 1$ such that in any time interval in which some processor takes $\Phi + 1$ steps, every nonfaulty processor must take at least one step.

Communication (c)

U *Asynchronous:* Messages can take arbitrarily but finitely many real time steps to be delivered. However, in any infinite run of the system, every message to a nonfaulty processor is eventually delivered: messages cannot be lost.

F *Synchronous:* There is a constant d such that every message is delivered within d real time steps.

Message Order (m)

U *Asynchronous:* Messages can be delivered out of order.

F *Synchronous:* If p sends m_1 to r at real time t_1, q sends m_2 to r at real time t_2, and $t_1 < t_2$, then r receives m_1 before m_2.

Transmission Mechanism (b)

U *Point-to-point:* In an atomic step, a processor can send to at most one processor.

F *Broadcast:* In an atomic step, a processor can broadcast a message to all processors. Since we place no bound on the length of messages, this includes sending a different message to each processor in an atomic step.

Receive/Send (s)

U *Separate:* In an atomic step, a processor cannot both receive and send messages.

F *Atomic:* Receive and send are part of the same atomic step.

Stockmeyer ([27]) has observed that an impossibility result for weak consensus in a "strong" model in which some parameter is favorable does not necessarily imply impossibility in the "weaker" model in which the parameter is unfavorable. There could be a trivial protocol, say, one that always decides v in the strong model, which is made nontrivial by the added indeterminism of the weaker model. However, as the more general treatment of this work in [6] shows, the intuition that there is a protocol in the weak model only if there is one in the strong model holds for our results.

3. Definitions

We cannot improve on the presentation in [6].

"In this section we extend the formal framework of Fischer, Lynch and Paterson [13] to handle our various system parameters. A *consensus protocol* is a system of N ($N \geq 2$) processors $P = \{p_1, \ldots, p_N\}$. The processors are modeled as infinite-state machines with state set Z. There are two special *initial states* z_0 and z_1. For $v = 0,1$ a processor is started in state z_v if its initial bit is v. Each processor then follows a deterministic protocol involving the receipt and sending of messages. The messages are drawn from an infinite set M. Each processor has a buffer for holding the messages that have been sent to it but not yet received. If message order is synchronous, each buffer is modeled as a fifo queue of messages. If message order is asynchronous, each buffer is modeled as an unordered set of messages. The collection of buffers support two operations:

Send(p,m): places message m in p's buffer;

Receive(p): deletes some collection (possibly empty) of messages from p's buffer and delivers these messages to p.

The exact details of what messages can or must be delivered by *Receive*(p) depend on the choice of system parameters and this defined precisely below.

First, consider cases where message order is synchronous. Each processor p is specified by a *state transition function* δ_p and a *sending function* β_p where

$$\delta_p : Z \times M^* \rightarrow Z$$
$$\beta_p : Z \times M^* \rightarrow \{B \subseteq P \times M \mid B \text{ is finite}\}$$

The pair (q,m) in the range of β_p means that p sends message m to processor q. Since we place no constraints on the message set M, we can assume that for each $p,q \in P$, $z \in Z$ and $\mu \in M^*$ there is at most one message m with $(q,m) \in \beta_p(z,\mu)$. It is also convenient to assume that a processor attaches its name and a sequence number to each message so that the same message m is never sent by two different processors nor at two different times.

If transmission is *point-to-point,* then $|\beta_p(z,\mu)| \leq 1$ for every p, z and μ. If transmission is *broadcast* then p can send messages to any number of processors in one step.

[The details for the case receive/send separate are ommitted.]

A *configuration* C consists of

(i) N states $state(p_i,C) \in Z$ for $1 \leq i \leq N$, specifying the current state of each processor, and

(ii) N strings $buff(p_i,C) \in M^*$ for $1 \leq i \leq N$, specifying the current contents of each buffer.

Initially, each state is either z_0 or z_1 as described above, and each buffer contains λ (the empty string).

An *event* is a pair (p,μ) where $p \in P$ and $\mu \in M^*$. Think of the event (p,μ) as the receipt of message string μ by p. Processor p is said to be the *agent* of the event (p,μ). We now define conditions under which an event can be *applied* to a configuration to yield a new configuration.

(1) [Ommitted.]

...

(2) If communication is asynchronous, (p,μ) is *applicable* to C only if $\mu \in M \bigcup \{\lambda\}$ and μ is a prefix of $buff(p,C)$.

(3) If communication is *synchronous*, (p,μ) is applicable to C only if μ is a prefix of $buff(p,C)$.

(4) If communication is *immediate*, (p,μ) is *applicable to* C only if $\mu = buff(p,C)$.

If the event $e = (p,\mu)$ is applicable to C, then the next configuration $e(C)$ is obtained as follows:

(a) p changes its state from $z = state(p,C)$ to $\delta_p(z,\mu)$ and the states of the other processors do not change,

(b) for all $(q,m) \in \beta_p(z,\mu)$, append m to the right end of $buff(q,C)$,.

(c) delete μ from the left end of $buff(p,C)$.

In the case of *asynchronous message order,* the main difference in the above definitions is that each buffer is modeled as an unordered finite set. Therefore in the discussion above, M^* is replaced by the set of finite subsets of M, $buff(p,C)$ is a finite subset of M, events have the form (p,μ) where μ is a finite subset of M, and the empty set $\emptyset$ takes the place of λ. Minor modifications to the definition of "applicable" and "next configuration" must also be made, and we leave these obvious modifications to the reader; for example, in the case of asynchronous communication, (p,μ) is applicable only if $\mu \subseteq buff(p,C)$ and $|\mu| \leq 1$.

To define synchronous processors and synchronous (but not immediate) communication and to define correctness of a protocol, we must consider sequences of events. A *schedule* is a finite or infinite sequence of events. A schedule $\sigma = \sigma_1\sigma_2 \cdots$ is *applicable* to an initial configuration I if:

(1) the events of σ can be applied in turn starting from I, i.e., σ_1 is applicable to I, σ_2 is applicable to $\sigma_1(I)$, etc.;

(2) if processors are Φ-synchronous (constant $\Phi \geq 1$) then for every consecutive subsequence τ of σ, if some processor takes Φ steps in τ and if the processor p takes no steps in τ, then p takes no steps in the portion of σ following τ (this says that once a processor fails it cannot restart at a later time);

(3) if communication is Δ-synchronous (constant $\Delta \geq 1$) then, for every j, if $\sigma_j = (p,\mu)$, if message m was sent to p by the event σ_i with $i \leq j - \Delta$,... and if none of the events σ_k with $i<k<j$ is the receipt of m by p, then m belongs to μ (this says that messages must be delivered within Δ real time steps).

If $\Phi = 1$ in (2), the processors are said to be [in *lock-step* synchrony].

If σ is finite, $\sigma(I)$ denotes the resulting configuration, which is said to be *reachable* from I. A configuration reachable from some initial configuration is said to be *accessible*. Henceforth, all configurations mentioned are assumed to be accessible... A schedule together with the associated sequence of configurations is called a *run*.

We assume that there are two disjoint sets of *decision states* Y_0 and Y_1, such that if a processor enters a state in Y_v ($v \in \{0,1\}$) then it must remain in states in Y_v. A configuration C has *decision value* v if $state(p,C) \in Y_v$ for some p. A consensus protocol is *partially correct* if

(1) no accessible configuration has more than one decision value, and

(2) for each $v \in \{0,1\}$, some accessible configuration has decision value v.

A processor p is *nonfaulty* in an infinite run if it takes infinitely many steps and is *faulty* otherwise. For $0 \leq t \leq N$, an infinite run is a *t-admissible run from I* if:

(1) the associated schedule is applicable to I,

(2) at most t processors are faulty, and

(3) all messages sent to nonfaulty processors are eventually received.

A run is a *deciding run* if every nonfaulty processor enters a decision state. A protocol is a *t-resilient protocol for the weak consensus problem* if it is partially correct and every [infinite] t-admissible run from every initial configuration is a deciding run. A protocol is a *t-resilient protocol for the strong consensus problem* if it is partially correct, every [infinite] t-admissible run from every initial configuration is a deciding run, and if I_v is the initial configuration in which all processors have initial value v then all deciding configurations reachable from I_v have decision value v.

For the purposes of our impossibility proofs we would also like to define when a schedule is applicable to a noninitial configuration C. In the case of synchronous processors or synchronous communication the definition must depend not only on C but also on the history of events that led to C. If C is reached from initial configuration I by schedule τ, then σ is *applicable* to C iff $\tau\sigma$ is applicable to I. (To be completely precise we should include the history τ as part of the configuration C. However, in all our impossibility proofs, the history is clear from context, so we say simply that σ is applicable to C rather than to (C,τ).)

For configuration C let $V(C)$ be the set of decision values of configurations reachable from C. Configuration C is *bivalent* if $V(C) = \{0,1\}$, or univalent otherwise. Univalent configurations are either 0-*valent* if $V(C) = \{0\}$, or 1-*valent* if $V(C) = \{1\}$. The following obvious fact is used in often in our proofs:

For $v = 0,1$, if C is v-valent and D is reachable from C then D is v-valent."

4. Boundaries

To obtain the strongest possible results we establish some further conventions. Whenever we assume synchronous processors in an impossibility result, we actually take $\Phi = 1$, i.e., the processors operate in rounds. Whenever we assume synchronous communication in an impossibility result, we actually assume communication to be immediate. In results giving a consensus protocol, our protocol actually solves the strong consensus problem, defined like the weak consensus problem with the additional condition that if all initial values are the same, say v, then v is the decision value.

Theorem 3.1: In the model with asynchronous communication and asynchronous message order, and the other three parameters favorable, there is no 1-resilient weak consensus protocol ($\overline{cm}$ in figure 3.1).

Proof: The fully general proof appears in [6]. Here we prove impossibility assuming lock-step synchronous rounds and a prescribed order in which the processors take steps within each round. We assume a protocol exists and derive a contradiction.

Let P be a completely synchronized collection of processors operating in rounds. For any configuration C in P, let $turn(C) = i$ if and only if it is p_i's turn to take a step when P

	$s = 0$				$s = 1$			
pc \ $m\,b$	00	01	11	10	00	01	11	10
00	0	0	n	0	0	0	n	0
01	0	0	n	0	1	n	n	1
11	n	n	n	n	n	n	n	n
10	0	0	n	n	0	0	n	n

0 corresponds to unfavorable setting of the parameter

1 corresponds to favorable setting of the parameter

Figure 3.1: Results in [6]: Maximum Resilience

is in C. An event $e = (p_i, m)$ is *applicable to* C only if $turn(C) = i$.

A run σ is p_k*-free* if it does not contain the receipt by any $p_j \neq p_k$ of a message sent by p_k during σ. If a protocol is totally correct despite a single fault, then for all configurations C and for each processor p_k, there exists a p_k-free finite deciding run applicable to C. The messages sent by p_k during σ cannot affect the decision value of $\sigma(C)$ if σ is p_k-free. Two schedules applicable to C are said to be p_k*-similar* if they agree on all events (p_i, m_i), for all $p_i \neq p_k$.

Initialization Lemma: There exists a bivalent initial configuration.

Proof: The proof is as in [13]. See Appendix A.

Lemma 3.1: Let E be a reachable configuration of P and let $j = turn(E)$. Let σ and σ' be p_j-similar, p_j-free fdr's applicable to E. Then the decisions reached in $\sigma(E)$ and $\sigma'(E)$ must be identical.

Proof: By definition of p_j-similar only p_j can distinguish between the two runs while they are taking place, but by definition of p_j-free no message sent by p_j during either σ or σ' can be received by any of the other processors until after the run has been completed and a decision has been reached.

Lemma 3.2: Let E be a reachable configuration of P and let $e = (p_j, m_j)$ be applicable to E. If $e(E)$ is i-valent then for every p_j-free fdr σ applicable to E, $\sigma(E)$ has decision value i.

Proof: There exists a p_j-free fdr σ', applicable to E, beginning with e, such that σ and σ' are p_j-similar. Since $e(E)$ is i-valent $\sigma'(E)$ has decision value i. The result follows by lemma 3.1.

A configuration C is *failure-free bivalent* if there are configurations D_0 and D_1 reachable from C by failure-free runs, such that D_v is v-valent, $v = 0,1$.

Bivalency Lemma ([6]): C is failure-free bivalent if and only if C is bivalent.

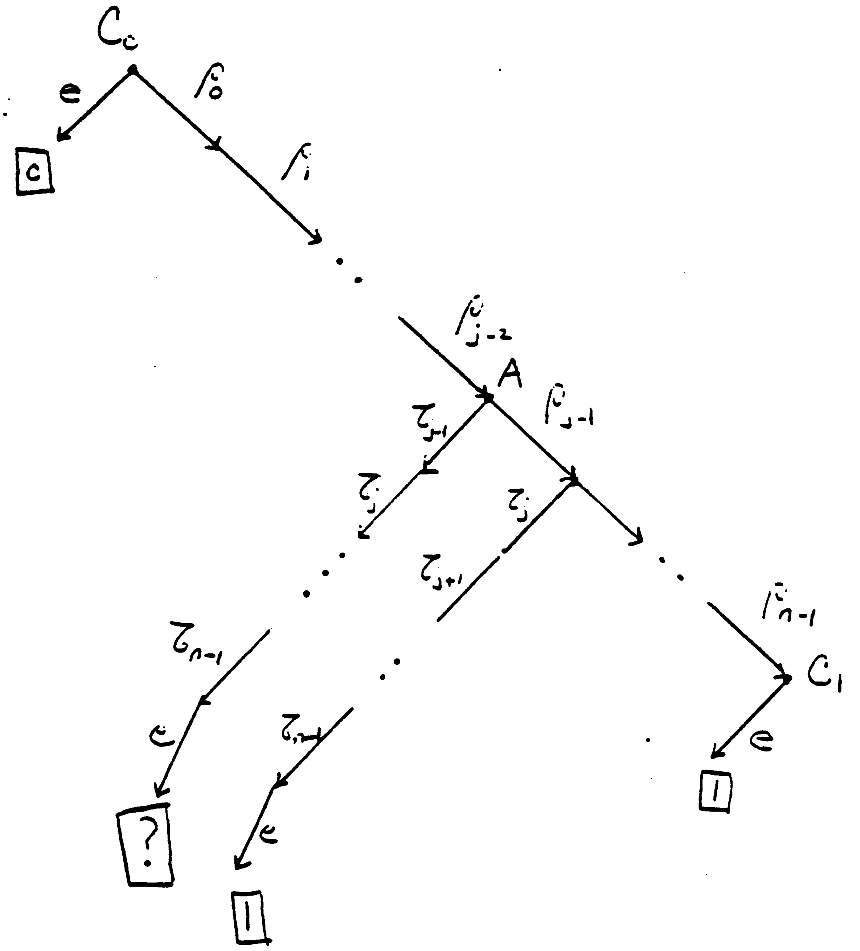
Lemma 3.3: Let C be a bivalent configuration of P, and let $e = (p,m)$ be an event applicable to C. Let Γ be the set of all configurations reachable from C_0 by a finite sequence of rounds, none containing e, and let Δ be the set $\{e(E) \mid E \in \Gamma\}$. Then Δ contains a bivalent configuration.

Proof: We assume every configuration in Δ is univalent and derive a contradiction.

Without loss of generality, let $turn(C) = 0$. Then $e = (p_0,m)$, for some message m. If p_0's message buffer is empty, then e is the only event applicable to C, whence, since C is failure-free bivalent by the bivalency lemma, $e(C)$ must be bivalent. If the buffer is not empty then, by a simple induction, there exist configurations C_0 and C_1, one reachable from the other by a single round, $\rho = \rho_0 ... \rho_{n-1}$, such that $e(C_i)$ is i-valent. Without loss of generality, let $C_1 = \rho(C_0)$.

For all j, $0 \leq j < n$, let τ_j denote the receipt of the empty message by p_j. That is, $\tau_j = (p_j, \emptyset)$. Let $T_j = \rho_0 ... \rho_{j-1} \tau_j ... \tau_{n-1}$. Then $T_{n-1} = \rho$, and any run associated with $[T_1 e]$ is p_0-free. By the assumption on Δ, $[T_1 e](C_0)$ is univalent, whence by the assumed 0-valence of $e(C_0)$ and an appliction of lemma 3.2, it must be 0-valent. However, we will prove by a backward induction on j that for all $1 \leq j < n$, $[T_j e](C_0)$ is 1-valent, a contradiction.

The basis $j = n-1$ is immediate, since $T_{n-1} = \rho$. Assume the statement holds for some T_j, $1 < j < n$, and consider the situation illustrated in figure 3.2. Let A be the configuration obtained by applying $\rho_0 ... \rho_{j-2}$ to C_0. Both $[\rho_{j-1} \tau_j ... \tau_{n-1} e]$ and $[\tau_{j-1} \tau_j ... \tau_{n-1} e]$ are the beginnings of p_{j-1}-free fdrs applicable to A. Applying either of them to A yields a univalent configuration, and by lemma 3.2 these configurations must have the same valence. Further, by the induction hypothesis, $[\rho_{j-1} \tau_j ... \tau_{n-1} e](A)$ is 1-valent, and therefore so is $[\tau_{j-1} \tau_j ... \tau_{n-1} e](A)$. But $[\tau_{j-1} ... \tau_{n-1} e](A)$ is precisely $[T_{j-1} e](C_0)$, so the induction holds and the desired contradiction has been obtained.



$\tau_0 = \rho_0$; $\mathbf{A}\ (i : 0 < i < n : \tau_i = (p_i, \emptyset))$

$T_j = \rho_0 \ldots \rho_{j-1} \tau_j \ldots \tau_{n-1}$; $A = [\rho_0 \ldots \rho_{j-2}](C_0)$

Figure 3.2

Using a slight modification of the technique of [13], given in Appendix B, we obtain an infinite, 0-admissible, nondeciding run of P. The constructions differ only in that the finite sequence added to the run at each step must be a sequence of rounds in our model, while in the construction in [13] it can be more general. We have actually proved impossibility of weak consensus even if the definition of the problem is further weakened to require only that even one processor decide. We have therefore strengthened the result of [13].

We can henceforth restrict attention to models where either communication or message order is synchronous. In each case we identify models where n-resilient protocols exist if and only if the model is not weakened. It is convenient to define a p_i-free schedule according to the intuitive notion suggested by the term. Henceforth, a schedule σ is p_i-free if p_i takes no steps during σ. The definition generalizes to sets of processors in the obvious way: if Q is any set of processors then σ Q-free if and only if for all $q \in Q$, σ is q-free. We first consider synchronous communication.

Theorem 3.2: If communication is synchronous, transmission is broadcast, receive/send is atomic, and the other two parameters are unfavorable, there is an n-resilient strong consensus protocol (cbs in figure 3.1).

Proof: The protocol for an arbitrary processor p_i is given in figure 3.3.

It remains to prove correctness. The guard on the loop is evaluated no more than $(2d+1)(n+1)+1$ times, as t can be reset to zero at most n times. Further, if execution of the loop is terminated byefore a decision is reached, then, since the cardinality of V_i is either one or two (there are only two possible decision values and p_i receives x_i within the first d iterations), one of the guards in the nested "if" will be satisfied, so a decision will be reached. We therefore have termination and decision for nonfaulty processors.

We now show consensus among decided processors. Let p_j be the first processor to send a "decide v" message (denoted $d(v)$). Assume for the sake of contradiction that some processor decides $\overline{v}$. Let p_i be the first such processor. Without loss of generality let p_j decide no later than p_i does. All times are on p_i's local clock.

If $d(v)$ is sent at or before $t_i - d$, then p_i receives it before making a decision, so it decides v. Thus $d(v)$ is sent at some time t_j, $t_i - d < t_j \leq t_i$. If $x \in V_j$ at t_j, then x is sent

```
{communication is d-synchronous (constant d ≥ 1)}
y_i, decided := b, false;
V_i := {x_i};
broadcast (i,x_i);
do (t ≤ 2d and ~decided) →
      receive (m);
      if m = "decide v" → y_i, decided := v, true
      [] m = (j,x_j) → V_i, t := V_i ∪ {x_j}, 0
      [] m = ∅ → t := t + 1
      fi
od;
if ~decided →
      if |V_i| = 1 → y_i := x_i
      [] |V_i| = 2 → y_i := 0
      fi;
        decided := true;
        broadcast "decide y_i"
fi
```

Figure 3.3: Protocol for p_i in the model with c, b, and s favorable

no later than t_j-2d, so $x \in V_i$ at t_i. If $x \in V_i$ at t_i, then x is sent no later than t_i-2d and therefore no later than t_j-d. Thus $x \in V_j$ at t_j. Since p_i and p_j follow the same algorithm, they must reach the same decision.

Remark: The protocol was contributed to [6] by Larry Stockmeyer.

Theorem 3.3 ([6]): If the model of Theorem 3.2 is weakened by making receive and send separate operations, then there is no 1-resilient weak consensus protocol ($\overline{pms}$ in figure 3.1).

Theorem 3.4: If the model of theorem 3.2 is weakened by having point-to-point transmission, there is a 1-resilient strong consensus protocol, but no 2-resilient weak consensus protocol. Moreover, this is true if message order is synchronous and if communication is immediate ($\bar{p}c\bar{b}s$ in figure 3.1).

Proof:

We first observe that protocol of figure 3.3 can be modified by initially setting $V_i := \{x_i\}$, $1 \leq i \leq N$, and replacing each broadcast by an $(n-1)$-cast to all processors except the sender (the proof is similar, but slightly harder). If $n = 2$ this is exactly point-to-point transmission. A 1-resilient protocol in the model with point-to-point transmission is therefore obtained by letting any two fixed processors reach a decision using the modified protocol. When one (or both) decides it sends the result to all other processors.

We now prove the impossibility result for two or more failures. Since all pending messages are delivered when a processor takes a step, an event can be specified by naming the agent. The flow of the proof is similar to that of theorem 3.1. We assume P is a 2-resilient protocol and derive a contradiction.

Lemma 3.4: There are no reachable configuration C_0, event p, and finite $\{p,y\}$-free schedule σ applicable to C_0, such that $\sigma(C_0)$ is i-valent and $\sigma(p(C_0))$ is $\sim i$-valent, where y is the processor written to by p at C_0, if one exists.

Proof: Assume the lemma is false for the sake of contradiction.

Without loss of generality, let $i = 0$. Let $C_1 = p(C_0)$ and let $\sigma(C_i)$ be i-valent, $i = 0,1$. Then **A** $x \notin \{p,y\}$,

$$state(x,C_0) = state(x,p(C_0)), \text{ and}$$
$$buff(x,C_0) = buff(x,p(C_0)).$$

Thus, any $\{p,y\}$-free fdr applicable to C_0 is applicable to $p(C_0)$, and the same decision must be reached in each case.

Lemma 3.5: Let C be a reachable configuration of P, and let $p(C)$ be i-valent. For any processor $q \neq p$, if $q(C)$ is univalent then it is i-valent.

Proof: If p writes to q at C, and q writes to p at C, then by two applications of lemma 3.4, if $p(C)$ and $q(C)$ are both univalent, then they have the same valence. If the mutual writing condition is not satisfied, then without loss of generality let p write to y at C, where $y \neq q$. Again by lemma 3.4, any $\{p,y\}$-free fdr applicable to C must have decision value i. But $q \notin \{p,y\}$ means that $q(C)$ is the first configuration reached in a $\{p,y\}$-free fdr from C, so if it is univalent, then it must be i-valent.

Lemma 3.6: Let C be a reachable bivalent configuration of P. Let p be an arbitrary processor, and let Γ be the set of configurations reachable from C without p taking a step. Let Δ be the set $\{p(E) | E \in \Gamma\}$. Then either Δ contains a bivalent configuration or for all q, $q \neq p$, there exists some configuration A, reachable from C by a finite schedule, such that $q(A)$ is bivalent.

Proof:

Let us assume every configuration in Δ is univalent. We will show that every processor $q \neq p$ can eventually take a step leaving the system in a bivalent configuration. Without loss of generality, let $p(C)$ be 0-valent. If for all other processors q, $q \neq p$, $q(C)$ is bivalent, then we are done. Otherwise, let q be some processor such that $q(C)$ is univalent. By lemma 3.5, $q(C)$ is 0-valent. By the bivalence of C there exists an fdr σ applicable to C, such that $\sigma(C)$ has decision value 1. Let A be the last configuration in the application of σ to C for which both $p(A)$ and $q(A)$ are 0-valent, and let $B = y(A)$ be the successor of A in σ (see figure 3.4).

Clearly $y \notin \{p,q\}$, since $\sigma(C)$ has decision value 1 while $p(A)$ and $q(A)$ are 0-valent. Now, y can write to at most one of p and q, so at least one of these processors receives exactly the same set of messages at A as it receives at B. Without loss of generality, let this

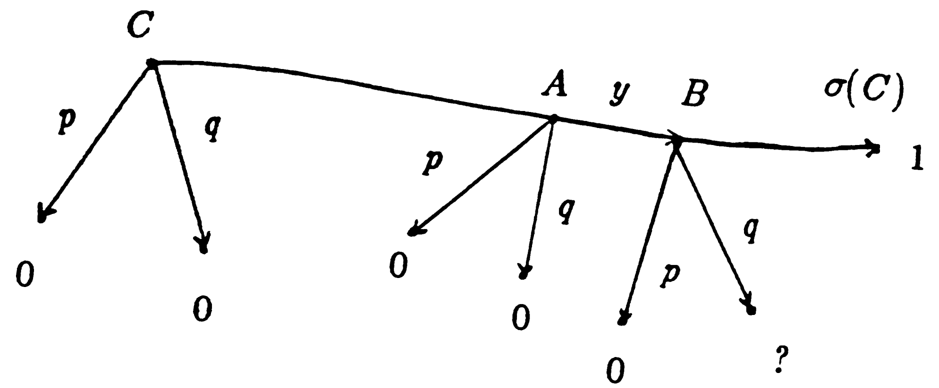


Figure 3.4

be true of p. By familiar arguments, $p(B)$ is 0-valent. By choice of A and B, $q(B)$ is bivalent or 1-valent. The latter case violates lemma 3.5, so $q(B)$ is bivalent.

A bivalent initial configuration is again obtained by the initialization lemma. We obtain an infinite, 1-admissible, nondeciding run by modifying the construction in [13] (see Appendix B), as follows. We maintain the processor queue as in the original construction, but when a processor takes a step it receives all pending messages. We assume inductively that the system is in a bivalent configuration, given initially by the initialization lemma. Let p head the queue, and let Γ and Δ be as in lemma 3.6. Then by that lemma, either Δ contains a bivalent configuration, in which case the construction proceeds as usual, or there is some way of moving any other processor such that the system remains in a bivalent configuration. If all n processors are in the queue, then p is removed and its successor becomes the new head. If some processor is already excluded from the queue then when p is removed, the previously excluded processor becomes the head. Either way, the lemma guarantees that the new head can move within a finite number of steps, leaving the system in a bivalent configuration. At most one processor is excluded from the queue at any time.

If a given processor is never excluded from the queue, then it takes an infinite number of steps, since any processor that is swapped in takes a step and then moves to the rear of the queue, allowing the given processor to advance in the queue and therefore to eventually take a step. If a processor is swapped out finitely often, there are two possibilities: after a finite number of steps it remains in forever or remains out forever. If it remains in, then it takes infinitely many steps. If it remains out, then the other $n-1$ processors each take infinitely many steps. Finally, if a processor is swapped out infinitely often, then it is swapped in infinitely often as well. Each time it is swapped in it takes a step, so it takes infinitely many steps.

Every time a processor takes a step it receives all undelivered messages sent at previous real times, so any processor taking infinitely many steps receives all messages sent to it. The run is therefore 1-admissible, infinite, and nondeciding. Theorem 3.4 is proved.

Theorem 3.5: If processors and communication are both synchronous (and the other three parameters are unfavorable), then there is an n-resilient strong consensus protocol (pc in figure 3.1).

Theorem 3.5 is well-known. In the model used in the chapter on distributed commitment, send and receive are split, there is no broadcast of any kind, and no assumptions are made on message delivery. The nonblocking commitment protocol described there can easily be modified to run with the weaker synchronization assumptions of the present model. However, the commit rule must be changed to logical OR in order to obtain *strong* consensus in the presence of failures.

Theorem 3.6 ([6]): If the model of theorem 3.5 is weakened by making processors asynchronous, there is no 1-resilient weak consensus protocol. Moreover, this is true even if message order is made synchronous. ($\overline{p}\overline{b}\overline{s}$ in figure 3.1).

In the next group of theorems, message order is synchronous.

Theorem 3.7: If message order is synchronous, transmission is broadcast and the other three parameters are unfavorable, then there is an n-resilient strong consensus protocol (mb in figure 3.1).

Theorem 3.8: If the model of theorem 3.7 is weakened by having point-to-point transmission, then there is no 1-resilient weak consensus protocol. Moreover, this is true even if receive/send is made atomic. ($\overline{p}\overline{c}\overline{b}$ in figure 3.1).

Theorems 3.7 and 3.8 are special cases, of theorem 3.10 and and the corollary to theorem 3.11, respectively, presented in the next section.

Theorem 3.9 ([6]): If message order and processors are both synchronous (and the other three parameters are unfavorable), then there is an n-resilient strong consensus protocol (pm in figure 3.1).

It follows from previous results that any weakening of the model of theorem 3.9 cannot tolerate even one failure. These theorems cover all 32 cases of choosing the five parameters.

5. A Second Type of Boundary

In the previous section we discuss only all or nothing propositions: the existence of a protocol resilient to one or two failures implied the existence of an n-resilient one. In practice, requiring a system to be n-resilient may be excessively stringent. We therefore examine k-resiliency, for arbitrary $1 \leq k \leq n$. Specifically, we assume processors and communication to be asynchronous and ask what is needed to obtain k-resilience.

Theorem 3.10: Serializable k-casting is a sufficient condition for $(k-1)$-resilience.

Proof: Let S be a k-element subset of the processors. Let each member of S broadcast its initial value to all of S (including itself) and then try to receive a message until one arrives. Serializability guarantees that all processors in S receive the same first message, and this is the decision value they adopt. Each processor in S then takes $\lceil \frac{n-k}{k} \rceil$ steps to broadcast the decision to $N - S$. The protocol is $(k-1)$-resilient: if $k-1$ processors fail at least one in S survives. Processors in $N - S$ can therefore try to receive until a decision message from some $p \in S$ arrives.

Corollary (Theorem 3.7): Serializable broadcasting suffices for n-resilience.

Proof: If at most $n - 1$ processors fail then by theorem 3.10 a decision is reached. If all n processors fail then the problem is solved trivially.

Serializable k-casting is also necessary for $(k-1)$-resilience in the sense that both serializable $(k-1)$-casting and "unserializable" k-casting are insufficient. Rather than prove two separate theorems, we prove a single, more general theorem of which these are special cases.

Let C be a configuration of P, and let events e_1 and e_2 applicable to C result in broadcasts of messages m_1 and m_2, respectively. If there is a set of processors Q, such that the processors in Q always receive the messages in the order in which they were sent, but for all $p \notin Q$ the order in which p receives the messages is arbitrary, then the broadcasts are said to be $|Q|$-*ordered.*

Theorem 3.11: $(k-1)$-ordered broadcasting is insufficient for $(k-1)$-resilience.

Lemma 3.7: Let C be a reachable bivalent configuration of P and let C_0 be reachable from C. Then there are no events $e = (p_e, m_e)$ and $f = (p_f, m_f)$ applicable to C_0 such that $e(C_0)$ is i-valent and $e(f(C_0))$ is $\sim i$-valent, $i = 0,1$, $p_e \neq p_f$.

Proof: Suppose such e and f exist. Without loss of generality, let $i = 0$. Let $C_1 = f(C_0)$. $D_i = e(C_i)$ is i-valent, $i = 0,1$. Now, $state(p_f, C_0) = state(p_f, D_0)$, so f is applicable to D_0, and p_f will behave the same way when f is applied to D_0 as when f is applied to C_0.

Let Q be the set of all processors that receive the messages from p_e and p_f in the order in which they were sent. Q contains at most $k-1$ processors. Further, since these messages are not ordered in the buffer of any processor not in Q,

$$\mathbf{A}\, r:\ state(r, [ef](C_0)) = state(r, [fe](C_1)), \text{ and}$$
$$\mathbf{A}\, r \notin Q:\ buff(r, [ef](C_0)) = buff(r, [fe](C_1)).$$

Let τ be any Q-free finite deciding run applicable to $e(D_0)$. No $r \notin Q$ can ever distinguish $e(D_0)$ from D_1, so τ is applicable to D_1 and the same decision must be reached in each case. Thus at least one of D_0 and D_1 must be bivalent, contradicting the assumption that D_i is i-valent, $i = 0,1$.

Lemma 3.8: Let C be a bivalent configuration of P, and let $e = (p, m)$ be an event applicable to C. Let Γ be the set of configurations reachable from C without applying e, and let Δ be the set $\{e(E) | E \in \Gamma\}$. Then Δ contains a bivalent configuration.

Proof: For the sake of contradiction we assume Δ only contains univalent configurations. By an easy induction there exist neighboring configurations C_0 and C_1 in Γ such that $D_i = e(C_i)$ is i-valent. Without loss of generality, let $C_1 = e'(C_0)$, for some event $e' = (p', m')$. If $p \neq p'$, then we have C_0 and C_1, C_1 reachable from C_0 by a single event e', and a p'-free event e such that $e(C_i)$ is i-valent, $i = 0,1$. This contradicts lemma 3.7, so $p = p'$.

If there exists a p-free fdr σ applicable to C_0 and a configuration E in the run $\sigma(C_0)$ such that $e(E)$ is 1-valent, then there exist neighboring configurations E_0 and E_1 in $\sigma(C_0)$ such that $e(E_i)$ is i-valent, contradicting lemma 3.7.

Let σ be any p-free, finite, deciding run applicable to C_0. We need only consider the case in which $A = \sigma(C_0)$ has decision value 0 and e applied to any configuration in the run is 0-valent. The schedule $[e' \, e]$ is applicable at any time during the run $\sigma(C_0)$, so there exist an event b and configurations B_1 and B_0 in the application of σ to (C_0) such that $B_0 = b(B_1)$ and $[e' \, e](B_i)$ is i-valent (see figure 3.5).

Let q be the agent in event b. Since σ is p-free, $q \neq p$. Let Q be the set of processors always receiving messages from p and q in the order in which they are sent when $[be']$ or $[e' \, b]$ is applied to B_1. Then $|Q| \leq k - 1$. The same Q-free finite deciding runs apply to $[be'](B_1)$ as apply to $[e' \, b](B_1)$. Unfortunately, e may not be a move in a Q-free schedule: p could be in Q. However, both e and e' are applicable to C_0, so neither event can involve

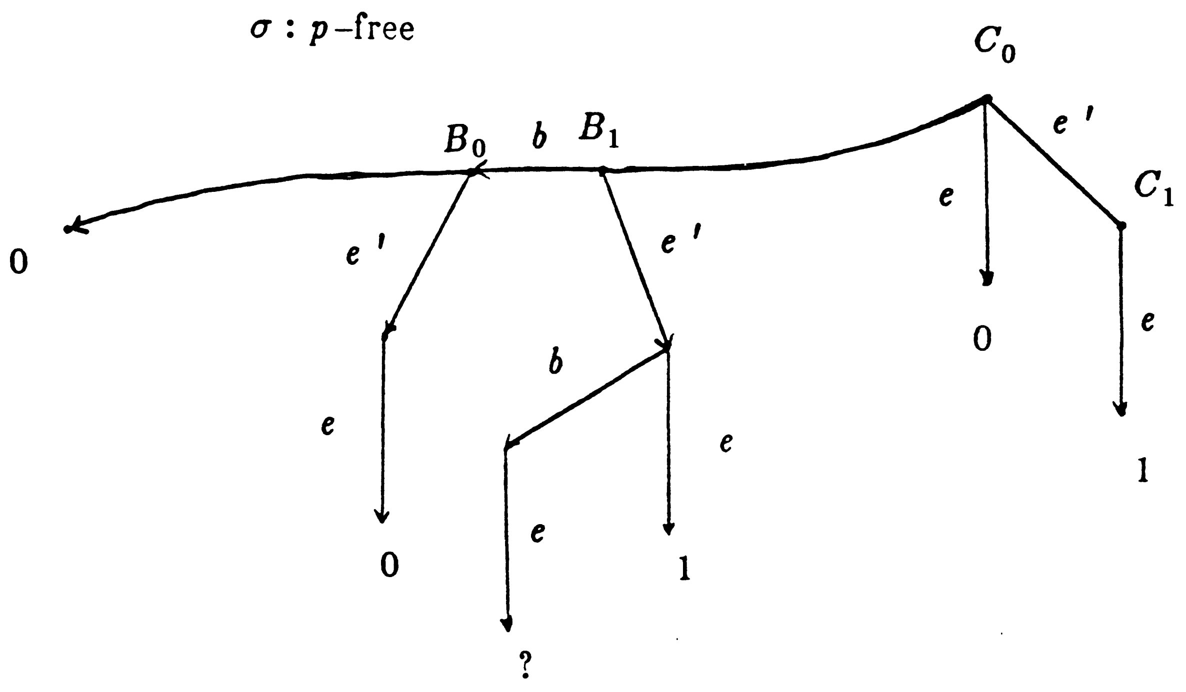


Figure 3.5

the receipt of a message sent by q when b is applied to B_1. Thus, p behaves identically when applying e to $[be']\,(B_1)$ as it does when applying e to $[e'\,b](B_1)$. Therefore, the same Q-free finite deciding runs are applicable to $[be'\,e](B_1)$ as to $[e'\,be](B_1)$, and the same decision must be reached in both cases, so the configuration $[e'\,be](B_1)$ is 0-valent. But $[e'\,e](B_1)$ is 1-valent, and by lemma 3.7 there can be no event b with agent $q \neq p$ such that $[e'\,be](B_1)$ is 0-valent and $[e'\,e](B_1)$ is 1-valent. Thus $[e'\,be](B_1)$ is 1-valent, a contradiction.

The construction of a non-deciding 0-admissible run is as in [13], as is the proof of the existence of a bivalent initial configuration. Theorem 3.11 is proved.

Corollary 3.1: Ordered $(k-1)$-casting is insufficient for $(k-1)$-resiliency.

Proof: Immediate from the theorem and the observation that any protocol in this model can be simulated by a protocol in a model with $(k-1)$-ordered broadcasting.

Theorem 3.8 is the special case $k = 2$ of corollary 3.1.

Corollary 3.2: 1-ordered broadcasting is insufficient for 1-resilience.

Proof: Let $k = 2$ in theorem 3.11.

The Fischer, Lynch, and Paterson result ([13]) can be restated as an easy consequence of corollary 3.2.

Corollary 3.3 ([FLP]): 0-ordered broadcasting is insufficient for 1-resilience.

CHAPTER 4

Commitment of a Distributed Transaction

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; it is useful both in the conceptualization and the realization of such systems. Historically, transactions have been associated with database systems, but the notion has broad applicability.

A distributed transaction (henceforth, a "transaction") may be viewed as a collection of single-processor *subtransactions*, coordinated by a *commitment protocol*. It is the responsibility of the commitment protocol to establish the atomicity of the transaction. Thus, a transaction is no more tolerant to processor failures than the commitment protocol coordinating its execution.

An extremely important class of fault-tolerant commitment protocols is the *nonblocking protocols*. A nonblocking protocol can terminate a transaction correctly so long as processor failures are not malicious and one of the participating processors remains operational. Hence, such protocols never "block": operational processors never suspend execution pending the recovery of a failed processor. In this sense nonblocking protocols are maximally tolerant of benign processor failures[1].

In spite of their higher 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 the total number of messages sent or maximum tandem message delay (i.e., the length of the longest

[1]A benign failure is one in which a processor ceases to take steps.

"path" of messages, corresponding roughly to circuit depth or time).

In this chapter we explore the inherent complexity differences between blocking and nonblocking protocols. The metrics are *messages* and *time*. Specifically, we study the cost of a "best-case" execution 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, 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 frustrated by our attempts to prove that this disparity was 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.

Finally, the bounds on nonblocking commitment protocols were found to apply to certain kinds of *broadcasts*, primitives for distributed systems.

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 constant Δ_p measured on the processor's local clock.

These rather strong assumptions are frequently assumed for many applications. Relaxing (3) requires solving a variant of the Byzantine General's problem at considerably more expense ([5], [7], [8], [18], [19], [20]). Relaxing (4) may make the problem unsolvable for even the two-processor case ([13]; see also chapter 3 of this thesis and [6]). 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 (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 (cf. [17]). For simplicity we will assume a completely synchronous system. In one (not necessarily atomic) *step* each processor can:

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

A processor need not take a step as an atomic action. 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. A processor *fails at time t* if it successfully completes step $t-1$ but does not complete step t. Failed processors simply halt; they do not recover during execution of a transaction.

2.2. Commitment Protocols

Transactions are decomposed into subtransactions, which are then distributed to participating processors for execution. Each processor is given the opportunity to vote ("commit" or "abort") 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, which is normally interpreted as an implicit vote to abort.

A *commit rule* governs when a transaction may be committed, and it is a function of the votes (implicit and explicit). A frequently used rule, assumed herein, is *unilateral abort*: any (implicit or explicit) vote to abort causes abortion.

A *commitment protocol* is a strong consensus protocol with the added constraint of the commit rule. The formal model for this chapter is almost identically the completely synchronous model of the previous chapter ($\Phi = \Delta = 1$), but with more parallelism. In this model, at each tick of the real time clock *every* processor takes exactly one step. For this reason we introduce the notion of a "failure event," taken only from a failed state f. For all buffer contents μ and all processors p, $\delta_p(f,\mu) = f$ and $\beta_p(f,\mu) = \emptyset$. A failure event affects no processor states or message buffers, not even those of the agent.

Formally, a *step* is a set of n events $(p_i,\mu_i) \in P \times M^*$, $0 \leq i < n$, where n is the number of processors in the system. A step $s = \{(p_i,\mu_i) \mid 0 \leq i < n\}$ is applicable to a configuration C only if $\mu_i = buff(p_i,C)$, $0 \leq i < n$. The resulting configuration $s(C)$ is obtained as follows:

(a) each p_i, $0 \leq i < n$, changes its state from $z_i = state(p_i,C)$ to $\delta_{p_i}(z_i,\mu_i)$;

(b) for all $(q,m) \in \bigcup(i: 0 \leq i < n: \beta_{p_i}(z_i,\mu_i))$ message m is added to $buff(q,C)$;

(c) μ_i is deleted from $buff(p_i,C)$ for all nonfaulty p_i, $0 \leq i < n$.

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 if no processor fails, transactions satisfying the commit rule are actually committed.

Let P be a commitment protocol and let s be a local state of some process p. The *concurrency set* of s, denoted $C(s)$, is the set of all local states s' such that s and s' appear in the same reachable global system configuration. If s is a commit state, then by *C1* $C(s)$ cannot contain an abort state. Similarly, if s is an abort state, then the concurrency set cannot contain a commit state.

To understand nonblocking protocols, we must consider the situation in which s is not a final state. (It will be convenient to assume that a processor's state is its initial state together with its entire history. While this assumption is not necessary, it simplifies the arguments immensely.) If s is not a final state, $C(s)$ contains neither an abort nor a commit state, and all processes but p fail, then p may decide whether to abort or commit according to the commit rule and other criteria, and consistency will be maintained trivially. On the other hand, if, without loss of generality, $C(s)$ contains a commit state, then it cannot also contain an abort state, for if all other processors fail, then p cannot decide without possibly introducing inconsistency.

Let P be a nonblocking commitment protocol. Then there exists a partition of the local states of P into two classes, *committable* and *noncommittable* satisfying the following conditions, where p is an arbitrary processor:

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

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

N3. Occupancy of a committable state implies satisfaction of the commit rule.

In fact, we may adopt the following classification. Local state s of processor p is *committable* if and only if $C(s)$ contains no abort state and p never enters s in any execution in which the commit rule is not satisfied. Otherwise it is *noncommittable*. (The partition idea first appears in [26], although there are errors in the existence proof. A more rigorous treatment and complete proof are in preparation [12].)

3. Message Complexity

Lemma 4.1: Let P be a blocking or nonblocking commitment protocol, and let p be an arbitrary processor in P. If I is a failure-free execution of P that results in commitment, then for every processor q, $q \neq p$, there is 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.

Proof: Let n be the number of processors in P and let I require time t. We construct the *message graph* $G = (V,E)$ corresponding to execution 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 the *message* edges E_m represent 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. Formally,

$$E_m = \bigcup (p,q\colon p \neq q \text{ and } p \text{ sends a message to } q \text{ at time } i\colon \{<(p,i),(q,i+1)>\}),$$
$$E_r = \bigcup (p,i{:}0 \leq i < t{:}\{<(p,i),(p,i+1)>\}).$$

The edges are directed. For any vertex $v \in V$, $D(v)$ denotes the subdag of G induced by v. For all processors p, we let $G(p) = D((p,0))$.

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 one input to each of the n outputs.

We prove in lemma 4.2 that for any processor p, $G(p)$ is an n-distributor, and this completes the proof of lemma 4.1.

Lemma 4.2: $G(p)$ is an n-distributor.

Proof:

For the sake of contradiction, let us assume that $G(p)$ is not an n-distributor. Then there is a nonempty set of processors Q satisfying

$$\mathbf{A}\ (q : q \in Q : (q,t) \notin G(p)).$$

We construct a "bad" execution B of P in which p fails before voting (an implicit vote to abort), but the processors in Q receive exactly the messages they receive in execution I. Each processor r fails in B at the least s such that $(r,s) \in G(p)$, and the processors in Q never fail. At any instant, the state of an operational processor in B is identical to its state at the corresponding instant of I, so it sends exactly the same messages in both executions. Thus, a nonfailed processor cannot distinguish the good execution from the bad on the basis of messages received or messages not received due to failure of the sender. Whether a processor has failed (in B) or has been reached by p (in I), it does not write to processors in Q. The processors in Q can never distinguish B from I, and therefore they commit in both executions, violating *C2* in B.

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

Corollary 4.1: Let P be a commitment protocol, and let I be a failure-free execution of P which results in commitment. Then the message graph corresponding to I is an (n,n)-distributor.

Proof: Immediate from n applications of lemma 4.2.

Corollary 4.2: $|E_m \cap G(p)| \geq n - 1$.

Proof: By lemma 4.2, $(r,t) \in G(p)$, for all processors r. Thus, if $r \neq p$, then there is at least one edge in $G(p)$ incident on row r and originating in some row q, $q \neq r$. But E_m is precisely the set of edges between distinct rows, so

$$|E_m \bigcap G(p)| \geq |\{r : r \neq p\}| = n - 1.$$

Lemma 4.3: 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

$$M = \bigcup (p_j : p_j \in S : E_m \bigcap G(p_j)).$$

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

Proof:

The proof is by induction on i, the cardinality of S. The basis, $i = 1$, is given by corollary 4.2.

For $i \geq 2$, assume the lemma holds for $i - 1$ and let S be as in the statement of the lemma. Then

$$\mathbf{E}\ (p,e : p \in S, e \in G(p) \bigcap E_m : \mathbf{A}\ (p' : p' \in S - \{p\} : e \notin G(p'))).$$

That is, there exist a processor $p \in S$ and a message edge e such that 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:

$$M' = \bigcup (p_j : p_j \in S' : E_m \bigcap G(p_j)).$$

Then $e \notin M'$. By the inductive hypothesis $|M'| \geq n + i - 3$. Since M' is properly contained in M,

$$|M| \geq |M'| + 1 \geq n + i - 2.$$

Theorem 4.1: Any commitment protocol 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 4.3.

Theorem 4.1 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 is too low for nonblocking protocols. This, however, is not the case.

Theorem 4.2: There exists a nonblocking commitment protocol requiring exactly $2n - 2$ messages in the absence of processor failures. The time required is $2n + 1$, independent of the number of failures.

Proof: Protocol COMMIT, which appears in figures 4.2a and 4.2b, achieves these bounds. Figure 4.2a contains the protocol observed by a distinguished processor p_0, figure 4.2b shows the protocol followed by all other processors, which is also represented schematically in figure 4.1. The subtransactions are assumed to have been distributed to the individual processors. (In section 5 these bounds are achieved without this assumption.) The protocol of the distinguished processor has two phases: *voting* and *reporting*. In the voting phase (step 0) processors send their votes to p_0, which does nothing until step 1. During step 1 p_0 receives the

step	possible message received	possible action	phase
0	$\emptyset$	send vote to p_0	vote
$i+1$	decision from p_0	send to p_1 for help	report
$i+2$	$\emptyset$	send to p_2 for help	report
$i+3$	decision from p_1	send to p_3 for help	report
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2i+1$	decision from p_{i-1}	decide	report
$2i+2$	*help* from p_{i+1}	send decision to p_{i+1}	confirm
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i+n$	*help* from p_{n-1}	send decision to p_{n-1}	confirm

Figure 4.1: Schematic Representation of Protocol for p_i, $i \neq 0$

votes, evaluates the commit rule, and decides ("commit" or "abort") accordingly. It then begins the reporting phase, sending the decision to each processor p_i at step i. In addition to the voting and reporting phases, the protocol for the general processor has a third phase, *confirmation*, during which processors ensure that the decision is sent to all operational processors, as shown in figure 4.1.

Lemma 4.4: If p_i has not failed by the end of step $2i + 1$, then y_i has a Boolean value, i.e p_i has decided whether to abort or to commit its subtransaction.

Proof: Immediate from the algorithm.

{the processors are $\{p_0,\ldots,p_{n-1}\}$}
{x_i is the Boolean vote of processor p_i, already initialized}
{y_i is the decision of p_i, initially ?, eventually Boolean}

step := 0;
do *step* $\leq n \rightarrow$
    if *step* $= 0 \rightarrow$ skip
    [] *step* $= 1 \rightarrow$
        receive messages sent during step 0 (if any);
        $y_0 := x_0$ & $n-1$ commit votes received;
        send y_0 to p_{step}
    [] $2 \leq$ *step* $< n \rightarrow$ send y_0 to p_{step}
    [] *step* $= n$ & $y_0 \rightarrow$ commit
    [] *step* $= n$ & $\overline{y_0} \rightarrow$ abort
    fi;
    step := *step* + 1
od

Figure 4.2a: Protocol COMMIT, Algorithm for p_0

```
{y_i = ? , x_i Boolean}
step := 0;
do step ≤ n + i →
     if step = 0 →
          if ¬x_i then y_i := false fi;
          send x_i to p_0
     [] 1 < step ≤ i → skip
     [] i < step ≤ n + i →
          possibly receive a message;
          if decision received then y_i := decision fi;
          if i+1 ≤ step < 2i & y_i = ?
               then send "help" to p_{step-i} fi;
          if step = 2i + 1 & y_i = ? then y_i := false fi;
          if "help" message received then send y_i to p_{step-i-1} fi
     fi;
     step := step + 1
od;
{step = n+i+1 and y_i is Boolean }
if y_i → commit
[] ¬y_i → abort
fi
```

Figure 4.2b: Protocol COMMIT: Algorithm for p_i, $i \neq 0$

Lemma 4.5: A processor only receives requests for help after it has decided of failed.

Proof: Fix any i and j, $0 < i < j < n$. 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 $2i + 2$, and by lemma 4.4 if p_i is operational then it has decided by the end of step $2i + 1$.

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.

Lemma 4.6 The algorithm runs in $2n + 1$ steps.

Proof: p_0 decides at step n and each p_i, $1 \leq i < n$ decides according to y_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.

Lemma 4.7: 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 sends a message to p_0. These processors send no further messages. For all i, $0 < i < n$, p_0 sends the decision to p_i. Thus each processor sends exactly one message except p_0, which sends $n-1$, for a total of $2(n-1)$ messages.

Our improvement over the conjectured lower bound is in the third phase, wherein explicit confirmation messages are omitted. This technique is used in chapter 2 ([3], [4], [23]), and a similar idea appears in [16].

4. Time Complexity

Information can be transmitted by a "nonmessage": in protocol COMMIT the absence 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 50% more time than the fastest blocking protocol. If notification is not included, then a blocking protocol can be about twice as fast as a nonblocking protocol.

Theorem 4.3: Let $n = 2^v$, for some integer v. Then there exists an n-processor nonblocking commitment protocol without notification which requires $2v-1$ steps to commit a transaction in the absence of failures.

$\{P$ is the set of all processors $\{p_0, \ldots, p_{n-1}\}$, $n = 2^v\}$

$\{x_i$ is the Boolean vote of processor p_i, already initialized$\}$

$\{y_i$ is the decision of p_i, initially ?, eventually Boolean$\}$

$\{z_i$ is the logical AND of the votes of processors which have reached $p_i\}$

$\{M_i$ is the set of processors which have reached $p_i\}$

$\{N_i$ is the set of processors known by p_i to know the decision or be failed$\}$

$step,\ z_i,\ M_i,\ N_i := 0,\ x_i,\ \{p_i\},\ \emptyset;$

$\{\text{I}::M_i = \{r \mid (i, step) \in G(r)\},\ z_i = \wedge(r : r \in M_i : x_r)\}$

do $step < v \rightarrow$

  if $i\ (mod\, 2^{step+1}) < 2^{step} \rightarrow j := i + 2^{step}$

  $[]$ $i\ (mod\, 2^{step+1}) \geq 2^{step} \rightarrow j := i - 2^{step}$

  fi;

  send $<M_i, z_i>$ to p_j;

  $step := step + 1;$

  try to receive a message from p_j;

  if message received **then** $M_i,\ z_i := M_i \bigcup M_j,\ z_i z_j$ **fi**

  $\{\text{I}\}$

  $\{p_j$ operational $=> M_j = M_i$ and $y_j = y_i\}$

  {in a failure-free execution

  $$M_i = \bigcup(k : 0 \leq k \leq step : \{p_r \mid \lfloor \frac{i}{2^k} \rfloor = \lfloor \frac{r}{2^k} \rfloor\})\}$$

od;

{in a failure-free execution $M_i = P$ and $z_i = \wedge(p_r :: x_r)\}$

Figure 4.3a: Protocol FAST_COMMIT, Voting and Reporting Phases Combined

Proof: Algorithm FAST_COMMIT, given in figures 4.3a, 4.3b, and 4.3c, runs in time $2v - 1$ and requires $2n \log n$ messages, in the absence of failures. The computation graph for a failure-free execution with $n = 8$ is shown in figure 4.4.

The algorithm has two phases in the absence of failures, with a possible third phase if failures occur. The communication graph of each of the first two phases is a butterfly network; the two networks share one level. Notice of a failure is propagated along the edges of the communication graph. Thus, notice of any failure in the first phase (e.g., some processor does not become committable) is propagated to all processors within the first $2\log n$ steps (and therefore before any site commits).

---

if $\overline{z_i} \rightarrow$ {commit rule not satisfied} $y_i,\ N_i :=$ false, $\{p_i, p_j\}$
▯ $z_i \rightarrow \{p_i$ committable or undecided, x_i true$\}$
    if $M_i = P$ **then** $y_i, N_i := true, \{p_i, p_j\}$ **fi**;
fi;
{if p_j operational, then $y_i = y_j$}
$\{$I2 :: in a failure-free execution $N_i = \{p_r : i \equiv r \bmod 2^{2v-step-1}\}\}$
do $step < 2v-1 \rightarrow$
    if $i \bmod 2^{2v-step-1} < 2^{2v-step-2} \rightarrow j := i + 2^{2v-step-2}$
    ▯ $i \bmod 2^{2v-step-1} \geq 2^{2v-step-2} \rightarrow j := i - 2^{2v-step-2}$
    fi;
    send $< N_i, y_i >$ to p_j;
    $step := step + 1$;
    try to receive a message from p_j;
    if message received $\rightarrow$
        $N_i := N_j \bigcup N_i$;
        if $y_j \neq ?\ \&\ y_i = ?$ **then** $y_i := y_j$ **fi**
    ▯ message not received $\rightarrow$ skip
    fi
    {I2}
od;
{in a failure-free execution $N_i = P$}
if $N_i = P \rightarrow$ decide according to y_i

Figure 4.3b: FAST_COMMIT, Confirmation Phase (p_i finished if it has committed)

The third phase works by brute force. Processors aware of a possible first phase failure and knowing the decision broadcast the decision, sending to each processor in turn and then entering a decision state. Any processor not knowing the decision after the third phase can conclude that all other processors are failed or undecided, and can therefore abort.

Corollary 4.3: There exists a nonblocking commitment protocol requiring $3\log_2 n - 1$ steps to commit a transaction in the absence of failures.

```
[] N_i ≠ P →
    do step < 2v + 2^v →
        if y_i ≠ ? → send y_i to p_{step-2v}
        [] y_i = ? and step=i+ 2v+ 1 →
            try to receive decision;
            if decision received then y_i := decision fi
        [] y_i = ? & step≠i+ 2v+ 1 → skip
        fi;
        step := step + 1;
    od
    if i = 2^v-1 & y_i = ?→
        try to receive;
        if decision received → y_i := decision
        [] decision not received → y_i := false
        fi
    fi;
    decide according to y_i
fi
```

Figure 4.3c: FAST_COMMIT, Extension of Confirmation Phase (failures have occurred)

Proof: The algorithm is obtained from algorithm FAST_COMMIT by prepending notification in the obvious way.

Lemma 4.8: Any commitment protocol of size n requires time $\log_2 n$ without notification and $2\log_2 n$ with notification.

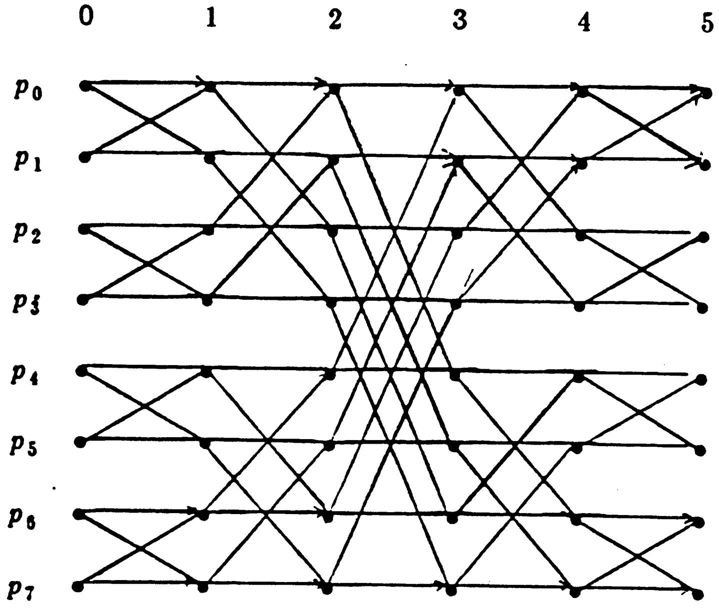


Figure 4.4: Failure-Free Execution of FAST_COMMIT when $n = 8$

Proof: Once a processor has been notified of the transaction it must explicitly reach each other processor, by lemma 4.1. Since the number reached at most doubles at each step the result for protocols without notification is obtained by an easy induction on time. If the processors have not been notified, then by the same argument there is at least one processor that is not notified before time $\log_2 n$, and hence cannot vote before that time. The $2\log_2 n$ bound follows from the first case.

In the following, all logarithms are taken base 2.

Theorem 4.4: Any nonblocking commitment protocol of size n requires at least $2\log n - 3\log\log n - O(1)$ steps without notification, $3\log n - 3\log\log n - O(1)$ steps with notification.

Proof:

We present the lower bound for a protocol with notification. The bound for protocols without notification is obtained by an almost identical argument. Let n be a power of 2.

Let P be a time-optimal nonblocking protocol for n processors, and let I be a failure-free execution of P resulting in commitment and requiring t steps. As before, x_i denotes the vote of processor p_i. This time, however, x_i is initially undefined for all but the initiator of the transaction (a processor cannot vote on the subtransaction before knowing what it is). Let v_i be the least time at which $x_i \neq ?$. We say a processor p_i is *precommittable* at time r if and only if $\mathbf{A}(j ::(p_i,r) \in D((p_j,v_j)))$. By the proof of lemma 4.2 and the choice of commit rule, a processor is committable only if it is precommittable.

Let t_1 be defined by

$$t_1 = \min\ (r : 0 \leq r \leq t : \mathbf{A}\ (p_i :: x_i \neq ?\ \text{at time } r\ )).$$

t_1 is the first step at which the votes of all processors are defined, so by step t_1 all processors have been notified of the transaction. Further, by choice of t_1, at least one x_i is undefined at t_1-1, so no processor can be precommittable at that time. Clearly, $t_1 \geq \log n$, while by corollary 4.3 $t < 3\log n$. Thus

$$t_2 = t - (t_1 - 1) \leq 2\log n\ .$$

Since I results in commitment, all n processors become precommittable during the last t_2 steps of the execution. In particular, there exists a step in which at least $\frac{n}{t_2}$ processors become precommittable. Let r be such a step, M the set of processors becoming precommittable at r, and S the complement of M.

Lemma 4.9: If $|M| \geq 3$, then the elements of M are not in commit states at the end of step r.

Proof: Let x and z be elements of M. Since z becomes precommittable 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, at step r, x can be certain that the message was sent only if it was sent by x. 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 precommittable at the end of step r and before step $r+1$.

By a similar argument, if $|S \bigcup M| > |M| \geq 2$ then no processor can commit before step $r+1$.

Let $k = t - r$. Since no processor is in a commit state at the end of step r, k steps suffice to move the elements of $S \bigcup M$ to commit states. At most $n(k+1)$ messages can be received in the $k+1$ steps $r, r+1, ..., r+k$. Suppose, for the sake of argument, that all these messages are received by processors in M. (This is absurd, but we err on the safe side). Then there exists some processor $p \in M$ that receives at most $\frac{n(k+1)}{|M|}$ messages during these $k+1$ steps. Fix such a processor p. Since $\frac{n}{|M|} \leq t_2$, p receives at most $t_2(k+1)$ messages during steps r through $r+k$.

We will construct an execution J in which p remains operational, but the processors sending to p at or after $r-1$ in I fail, as do certain others, and p does not become precommittable on schedule. Let G be the message graph corresponding to execution I, as in the proof of lemma 4.1. For all vertices $v \in G$ let $D(v)$ denote the directed, acyclic subgraph of G rooted at v. We are interested in a subgraph $FGH(p)$ induced by certain vertices in columns $r-1$ through t of G. The definition of $FGH(p)$ is slippery. To develop some intuition for what will be involved, consider the question: which processors can distinguish I from J? Our strategy will be to kill all processors but p as soon as they can distinguish the two executions. By time t, all but p must be dead, as otherwise they will commit, erroneously assuming p to be committable. It is important that p remain operational, as otherwise there will be no violation of nonblocking condition *N1*. There are three groups of processors to consider. One group contains p and the processors written to by p in J, together with the processors reached by these during I. Then there are the processors which write to p at or after $r-1$ of I, together with any processors they reach during I. Finally, there are the processors that p writes to during I but p does *not* write to during J, together with the processors reached directly or indirectly by these processors during I. These groups correspond, roughly, to subgraphs $F(p)$, $H(p)$, and $G(p)$, respectively, defined below.

$G(p) = D((p,r))$.

$H(p)$ denotes the subgraph of G induced by the processors sending to p at or after step $r-1$ of I:

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

$F(p)$ is the subgraph of G induced by processors sent to by p in J:

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

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

Notation: Let $FGH(p)$ denote the subgraph of G induced by the union of $F(p)$, $G(p)$, and $H(p)$:

$$v \in FGH(p) \text{ iff } v \in F(p) \bigcup G(p) \bigcup H(p)$$
$$<v,w> \in FGH(p) \text{ iff } <v,w> \in G \ \&\ v \in FGH(p)$$

Specification of J: Each processor x, $x \neq p$, fails at step $\mu i \ \{(x,i) \in FGH(p)\}$, denoted $fail(x)$.

Lemma 4.10: 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.

Proof: The initial state of x and the messages it receives in steps 0 through i are identical in the two executions.

Corollary 4.4: For all processors x, $(x,t) \in FGH(p)$.

Proof: Suppose not. Let q be any processor such that $(q,t) \notin FGH(p)$. q commits its subtransaction by step t of I. By lemma 4.10, if $(q,t) \notin FGH(p)$ then q still cannot distinguish J from I at t. Thus q commits in J as well, violating rule $N1$.

Let $\alpha = \max\,(x : x \neq p : fail(x)) - r$, 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 execution resulting in commitment requires time at least $r + \alpha$. Since $t = r + k$ we have $k \geq \alpha$.

Let the function $f(i)$ satisfy:

$$f(i) \geq |\{x : (x, r+i) \in FGH(p)\}|.$$

At the end of step r executions I and J differ only in the states of p and processors sending to p at $r-1$ and r. Let $d = t_2(k+1)$. Since p receives at most d messages during steps r through $r+k$ of I, at most d processors can send to p at or after step $r-1$, so at most d can possibly write to p at steps $r-1$ and r. 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 at most $f(0)$ 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 executions 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 $S \bigcup M = \{x :: (x, r+\alpha) \in FGH(p)\}$ satisfies

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

$$=> 2^\alpha(d+2) > n$$

$$=> 2^\alpha > \frac{n}{t_2(k+1)+2}$$

$$=> 2^\alpha \geq \frac{n}{t_2} / (k+1+\frac{2}{t_2})$$

$$=> \alpha \geq \log \frac{n}{t_2} - \log(k+1+\frac{2}{t_2}).$$

But in general, $\frac{2}{t_2} < 1$, and $k \leq t_2 \leq 2\log n$, whence

$$(1) \qquad \alpha \sim \log \frac{n}{t_2} - \text{loglog}\, n - O(1).$$

How large is r? For some p_i the value of x_i is undefined until step t_1. Fix such a p_i. Then

$$\mathbf{A}(x : x \in M : (x,r) \in D((p_i,t_1)))$$

so $r \geq t_1 + \log \frac{n}{t_2}$, and

$$(2) \qquad t_1 + \log \frac{n}{t_2} + \alpha \leq r + \alpha \leq r + k = t.$$

Using (1) as an approximation to α we rewrite (2) to obtain

$$t_1 + 2\log \frac{n}{t_2} - \text{loglog}\, n - O(1) \leq t,$$

whence, since $t_2 \leq 2\log n$ and $t_1 \geq \log n$,

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

Remark: Much of the material in sections 1 through 4 appears in [11].

5. Broadcast Protocols

We now consider a degenerate version of commitment with notification, in which processors are not given the opportunity to vote. Here the problem is for a distinguished processor, called the *source*, to broadcast a message to all nonfaulty processors. Initially, processors do not expect to receive a message. The role of processors that have received the message is not completely passive: if failures occur, these processors are responsible for sending the message to all remaining nonfaulty processors.

5.1. Reliable Broadcast

Reliable broadcast may be viewed as a variant of the problem of distributed consensus: operational processors must agree on whether the source sent a message. Further, once a processor has received the message it is *committed*; hearing the message is an irreversible act.

Formally, a protocol for reliable broadcast must establish the condition

$$RB = \mathbf{A}\ (p : \text{failed}(p) \ \vee \ \text{commit}(p)) \ \vee \ \mathbf{A}\ (p : \text{failed}(p) \ \vee \ \sim\text{commit}(p)).$$

Theorem 4.5: Let P be a reliable broadcast protocol for a set N of n processors. Any failure-free execution of P resulting in commitment requires at least $2n - 3$ messages.

Proof: Let I be an execution of P satisfying the conditions of the theorem, and let p be the source of the broadcast. To simplify the argument we assume p receives an initial message from some external source. Then every processor in N must receive at least one message. Further, since processors do not initially know there is a message being broadcast they can never notice its absence. Hence, processors that have received the message need explicit confirmation of the receipt of the message by others. Let z receive the message no sooner than any other processor. Then every processor q, $q \neq z$, must either write to z or receive explicit confirmation that the message has been sent to z. Implicit confirmation is clearly insufficient: z cannot send an implicit message if it does not know there is a broadcast protocol in progress, while an implicit message from any processor other than z is meaningless if the sender fails. By choice of z this confirmation can not be obtained from the first message received by q, since z was not informed before q. Thus q must receive at least two messages or receive one and write to z.

Let $G = (V,E)$ be a directed multigraph with n vertices, one for each processor and named accordingly. G has a directed edge (x,y) for every message sent by processor x to processor y. For all processors w distinct from z, vertex w either has indegree at least 2 or has indegree 1 and $(w,z) \in E$. Thus $|E| \geq 2(n-1)$. Finally, we subtract one message, as, contrary to our assumption, the source processor does not initially receive a message. The total number of messages is therefore at least $2n - 3$, as was to be shown.

Theorem 4.6: Reliable broadcast requires at most $2n-3$ messages in the absence of failures.

Proof: Figure 4.5 shows a failure-free execution of a reliable broadcast protocol requiring $2(n-1)$ messages. The broadcast originates at p_0 at time 0 and the message is sent to p_i, $1 \leq i < n$, at time $i-1$. p_i receives the message at time i, and commits at that time. If p_i,

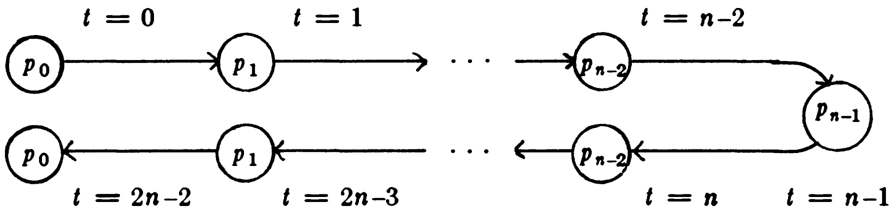


Figure 4.5: Failure-Free Execution of a Reliable Broadcast Protocol

$i < n-2$, does not receive its anticipated second message at $2n - 2 - i$, it begins a polling procedure similar to the one in protocol COMMIT, sending the message to each of $p_{n-1},\ldots,p_{i+2}$ in turn. Finally, the edge (p_{n-1},p_{n-2}) is unnecessary: p_{n-2} sent the message to p_{n-1} so it knows that if p_{n-1} is operational at $n-1$, then it will receive the broadcast at that time. Removal of this edge yields a total of $2n-3$ messages in the absence of failures. Correctness follows by an argument analogous to the proof of theorem 4.2.

Viewed differently, the communication scheme of figure 4.5 illustrates a nonblocking protocol *with notification* requiring $2(n-1)$ messages in the absence of failures. The notification and voting phases are combined in phase 1. The transaction is initiated at p_0. Each processor has a chance to abort the transaction unilaterally during the first steps, simply "rejecting" the transaction and sending it back rather than forwarding it. If some p_i does not receive its second message at $2n - i - 2$, it polls p_{n-1}, p_{n-2} and so on, in decreasing order. For any j, $i < j < n$, if p_j was reached in the first phase and remains operational, then it will know the decision by the time it is polled by p_i. If it was not reached in phase 1, then some processor failed before voting, whence the commit rule is not satisfied.

5.2. Atomic Broadcast:

Occasionally, reliable broadcast is insufficient, e.g. if commitment requires some action and if it is unacceptable for failed processors to have committed if the operational ones do not also commit. In this case the problem is to perform an *atomic broadcast.* The atomic broadcast condition is formally defined by

$$AB = \mathbf{A}\,(p\text{: failed}(p)\ \text{v}\ \text{committed(p))}\ \text{v}\ \mathbf{A}\,(p\text{: }\sim\text{commit}(p)).$$

Theorem 4.7: Atomic broadcast requires at least $2n - 3$ messages, and this number is sufficient.

Proof: The lower bound is an immediate corollary to theorem 4.5. This is because AB implies RB, so any protocol to establish AB will establish RB. However, the converse is false.

For this reason atomic broadcast was thought to be more expensive than reliable broadcast. In theorem 4.8 we show that any protocol for reliable broadcast can be trivially modified to yield a protocol for atomic broadcast. For each processor, the transformation will affect neither the number of messages sent nor the number of steps taken. Theorem 4.7 then follows from theorems 4.6 and 4.8.

Theorem 4.8: Atomic broadcast and reliable broadcast have exactly the same computational complexity.

Proof:

The crux of the broadcast problem is to maintain *consistency* without blocking, and this we know how to do. The precise definition of consistency will vary according to the desired outcome of a given protocol.

In analogy to the subtransaction processing states in a commitment protocol, let us define processing states for participants in a reliable broadcast protocol. We partition the states into three classes: *noncommittable, nonterminal commit,* and *terminal commit.* Processors are initially in noncommittable states. Occupancy of either type of commit state implies receipt of the message (for simplicity we assume the source has received the message). Occupancy of a terminal commit state implies satisfaction of *RB.* A processor in such a state has completed its role in the protocol.

If any two nonfaulty processors p and q occupy terminal commit and noncommittable states, respectively, then the system is said to be in an *inconsistent* state. But by definition, occupancy of a terminal commit state implies satisfaction of *RB,* and consistency must be maintained. Thus a processor may enter a terminal commit state only if all nonfailed processors are in commit states. For total correctness we require that any initiated broadcast actually be committed by all processors if there are no failures.

Let P be any protocol for reliable broadcast. Let Q be obtained from P as follows.

(1) Noncommittable states in P become noncommittable states in Q.

(2) Nonterminal commit states in P become *prepared to commit* states in Q, *not* commit states.

(3) Terminal commit states in P become *commit* states in Q.

The consistency requirement in P translates directly to requiring that no processor be in a commit state if another is in a noncommittable state. Q therefore "inherits" preservation of consistency from P, according to the original definition of the term (cf. section 2.2, this chapter). We can define a trivial commit rule, satisfied if and only if the source initiates a broadcast. Clearly, the broadcast will be committed only if it is initiated. Q therefore satisfies partial correctness conditions *C1* and *C2* for commitment protocols. For total correctness we require that a broadcast satisfying the commit rule, (i.e., an initiated broadcast), actually be committed if there are no processor failures. Total correctness for Q is inherited from P, since, in the absence of failures, all processors in P enter terminal commit states.

By construction of Q, occupancy of the prepared to commit state implies receipt of the broadcast message, which in turn implies satisfaction of the commit rule. Thus, since there are no abort states in P or Q, all states in Q other than the noncommittable ones are truly *committable* states, according to the original definition of the term (cf. section 2.2).

Q also satisfies the nonblocking conditions *N1* and *N2*. The second is trivial because there are no *abort* states. To obtain *N1*, observe that entering a commit state in Q corresponds to entering a terminal commit state in P, which in that protocol implies satisfaction of RB. If RB is satisfied in P, then in Q either all nonfaulty processors are noncommittable or all nonfaulty processors are committable. Thus processors in Q enter commit states only if all nonfailed processors are committable.

We have shown that Q satisfies all the requirements of a nonblocking commitment protocol, and therefore its effects are *atomic*. By construction, it is also a broadcast protocol. The theorem is proved.

In view of the last result it is not surprising that minor modifications to the proof of theorem 4.4 yield the same minimum time bound for both reliable and atomic broadcast as for nonblocking commitment without notification.

CHAPTER 5

Matrix Transposition Networks

1. Introduction

An n-matrix transposition network is a directed, acyclic, graph with n inputs and n outputs. Associated with each input is a set of n bits, corresponding to a row in an $n \times n$ Boolean matrix. Information (inputs or data computed from the inputs) flows from inputs to outputs so that eventually each output contains a row of the transpose of the original matrix.

If the information flows through the network according to prescribed paths, independent of the values of the input bits, the network is said to be *oblivious*. Only oblivious networks are considered in this chapter. If the bits are treated as *tokens* (atomic objects that cannot lose their identity) the network is said to be *conservative*. We prove that a conservative oblivious n-matrix transposition network of depth k requires $\theta(kn^{1+1/k})$ edges, for all $k \leq \log_2 n$.

The bounds apply to nonconservative networks of depth $k \leq 3$. We conjecture the lower bound holds for nonconservative networks of arbitrary depth, but the question is open.

2. Definition of the Model and Statement of the Problem

Let S be a fully connected distributed system consisting of a finite number of memoryless, minimally intelligent registers, and a set of tokens, initially distributed among certain distinguished registers called *inputs*. Communication between registers is restricted to passing nonempty packets of tokens, and occurs in lock-step synchrony. Packet delivery is instantaneous. A *step* of the system is the parallel execution by every register of:

Receive all packets sent at the previous step;
Unpack, permute, and repack all tokens into new packets, one for each intended receiver;
Send all new packets.

There is a unit cost associated with each packet sent, independent of the number of tokens it contains. No other messages are permitted. The tokens in the input registers are considered to have been sent at step -1.

In addition to the input registers we may designate a set of *output* registers, not necessarily disjoint from the inputs. A *protocol* for the system is a finite sequence of steps beginning with every token in an input register and ending with every token in an output register.

Let S have m registers, n inputs and n outputs, the latter two sets numbered from 0 to $n-1$. With each input let us associate an ordered set of n tokens, viewed as a row of an $n \times n$ matrix. The *n-matrix transposition* problem is to route the i^{th} token of each input register to the i^{th} output register. We wish to minimize the total number of packets sent as well as the number of steps required. The problem is made nontrivial by assuming that registers have a maximum capacity of n tokens. An *n-transposition protocol* for the system S is an oblivious, conservative routing scheme solving the matrix transposition problem.

Let P be any k-step protocol for S. We define the *message graph* $G(P) = (V,E)$, corresponding to P in a natural way. The vertex set, V, is a grid of $k+1$ columns and m rows. A vertex is denoted by its grid coordinates. Column 0 represents the processors in their initial state, and in general column i represents the processors at time i ($i \leq k$). The edges of G represent the set of packets sent during execution of P, and are directed from sender to receiver. Specifically,

$$E = \{<(r,i),(q,i+1)>: r \text{ sends to } q \text{ at time } i\}.$$

A register may send a packet to itself. The *weight* of an edge is the number of tokens in the corresponding packet. The weight of a vertex v, denoted $w(v)$, is the sum of the weights of its incoming edges. If $v = (r,0)$ then $w(v)$ is the number of tokens initially contained in register r.

If G is the message graph of a k-step matrix transposition protocol then the subgraph of G induced by the paths of the tokens is an oblivious, conservative, n-matrix transposition network of depth k.

3. Lower Bounds for Conservative Networks

For technical reasons we first study an extremely simplified version of the transposition problem. Let S be a system of $m \geq n$ registers with input set $\{a\}$ and outputs $\{b_0, \ldots, b_{n-1}\}$. Initially the input holds an ordered set of n tokens $x_0, \ldots, x_{n-1}$. The *distribution* problem is to send x_i to output b_i, for $0 \leq i < n$.

If D is the message graph for any protocol solving the distribution problem we say D is an *n-distributor*. Let $p(a, b_j)$ denote the path taken by x_j in D. For each vertex v in D let $d(v)$ denote the outdegree of v, and let $\mu(j,v)$ be the characteristic function with value 1 if and only if x_j passes through v on its way from a to b_j. That is,

$$\mu(j,v) = \begin{cases} 1 & \text{if } v \text{ is on the path } p(a,b_j) \\ 0 & \text{if otherwise.} \end{cases}$$

The number of tokens that pass through vertex v is given by $\sum_{0 \leq j < n} \mu(j,v)$. This is precisely the weight of v, so we have,

$$w(v) = \sum_{0 \leq j < n} \mu(j,v).$$

This quantity is bounded above by n, the maximum capacity of a register.

It is useful to measure how much of the distribution problem is solved by a given vertex, i.e., the progress made when tokens pass through the vertex. Intuitively, we see that a vertex requires large fanout in order to do a lot of distributing. It also must receive sufficiently many tokens to use its fanout in an interesting fashion. These considerations suggest a distribution measure which is the product of the outdegree of a vertex and its weight. Formally, for each vertex v in D the distribution measure of v, denoted $\delta(v)$, is defined as

$$\delta(v) = d(v)\, w(v) = d(v) \sum_{0 \le j < n} \mu(j,v).$$

We define the related quantity, $\Delta(D)$, to be the sum of the distribution measures of the vertices of D. That is,

$$\begin{aligned}\Delta(D) &= \sum_{v \in D} d(v)\, w(v) \\ &= \sum_{v \in D} d(v) \sum_{0 \le j < n} \mu(j,v).\end{aligned}$$

Since the quantity $d(v)w(v)$ is defined for vertices of arbitrary message graphs Δ is well defined for any message graph.

Lemma 5.1 yields a lower bound for $\Delta(D)$ in terms of n and the depth of D.

Lemma 5.1: If D is an n-distributor of depth at most k then $\Delta(D) \ge kn^{1+1/k}$.

Proof:

Pippenger and Yao [21] have shown that if T is a *tree* with n leaves and depth at most k then $\Delta(T) \ge kn^{1+1/k}$. Suppose D is not a tree. Let v be an arbitrary vertex in D with indegree $p \ge 2$. D may be modified by creating, for each edge e_i into v, a new vertex v_i, with unique inedge e_i, but with a copy of the outedges of v. The modification clearly does not affect the depth of the dag, and

$$\begin{aligned}\sum_{1 \le i \le p} \delta(v_i) &= \sum_i d(v_i)\, w(v_i) \\ &= \sum_i d(v)\, w(v_i) \\ &= d(v) \sum_i w(v_i) \\ &= d(v)\, w(v) = \delta(v),\end{aligned}$$

so $\Delta(D)$ is unchanged.

An easy induction on the depth of D shows that D can be unfolded to form a tree T of the same depth as D, with $\Delta(D) = \Delta(T)$, and such that T has at least as many outputs as

D.

It follows from the result of Pippenger and Yao that $\Delta(D) = \Delta(T) \geq kn^{1+1/k}$, as was to be shown.

Theorem 5.1: Let $N = (V,E)$ be an n-matrix transposition network of depth k. Then $|E| \geq \Omega(kn^{1+1/k})$.

Proof:

The argument is a modification of the proof of an analogous result in [21] for shifting networks. For each i, $0 \leq i < n$, let D_i denote the subgraph of N rooted at input a_i. D_i is an n-distributor, as there must be a path from a_i to every output. Let x_{ij} be the j^{th} token initially at input a_i, and let the definition of μ be extended to

$$\mu(i,j,v) = \begin{cases} 1 & \text{if } v \text{ is on the path taken by } x_{ij} \text{ from } a_i \text{ to } b_j \\ 0 & \text{if otherwise} \end{cases}$$

The weight of v is now given by

$$w(v) = \sum_i \sum_j \mu(i,j,v).$$

We obtain a lower bound on $\Delta(N)$ by n applications of lemma 5.1. Thus,

$$\begin{aligned}
\Delta(N) &= \sum_{v \in N} d(v)\, w(v) \\
&= \sum_{v \in N} d(v) \sum_{0 \leq i < n} \sum_{0 \leq j < n} \mu(i,j,v) \\
&= \sum_{0 \leq i < n} \sum_{v \in N} d(v) \sum_{0 \leq j < n} \mu(i,j,v) \\
&\geq \sum_{0 \leq i < n} \sum_{v \in D_i} d(v) \sum_{0 \leq j < n} \mu(i,j,v) \\
&= \sum_{0 \leq i < n} \Delta(D_i) \\
&\geq kn^{2+1/k}, \text{ by lemma 5.1.} \quad (*)
\end{aligned}$$

By contrast, since $w(v) \leq n$, we have

$$\begin{aligned}\Delta(N) &= \sum_{v \in N} d(v)\, w(v) \\ &\leq \sum_{v \in N} d(v)\, n \\ &= n \sum_{v \in N} d(v) \\ &= n\,|E|.\end{aligned}$$

Combining this with (*) yields

$$\begin{aligned}& n\,|E| \geq \Delta(N) \geq kn^{2+1/k} \\ & \Rightarrow |E| \geq kn^{1+1/k}.\end{aligned}$$

If the different levels of vertices represent the same physical set of registers, and if edges between copies of the same register are deleted, $|E|$ is at least $kn^{1+1/k} - kn = kn(n^{1/k} - 1)$.

4. Upper Bounds

We now show that the bounds of the previous section are tight. Let $r \geq 2$ be an integer. An r-ary butterfly network of depth t, denoted B(r,t), is defined recursively for all $t > 0$ as follows:

$t = 1$: $B(r,1) = K_{r,r}$, the complete bipartite graph on r vertices, with all edges directed from inputs to outputs.

$t > 1$: B(r,t) is constructed from r copies of B(r,t-1). It has r^t inputs and outputs, both sets numbered from 0 to $r^t - 1$. For all i, $0 \leq i < r^t$, there is an edge directed from input i to input i mod (r^{t-1}) of each copy of $B(r,t-1)$. Thus, the inputs of $B(r,t)$ all have outdegree r, and each of the inputs of the r copies of $B(r,t-1)$ has indegree r.

Let $|G|$ denote the number of edges in any graph G.

Lemma 5.2: $|B(r,t)| = tr^{t+1}$.

Proof: The lemma is immediate for $t = 1$. Assume the lemma inductively for $t-1$, where $t > 1$. Since each input of B(r,t) has outdegree r and there are r^t inputs, we have

$$\begin{aligned} |B(r,t)| &= (r^t)r + r\,|B(r,t\text{-}1)| \\ &= r^{t+1} + r((t-1)r^t) \\ &= tr^{t+1}. \end{aligned}$$

Let the vertices of B(r,t) have capacity r^t.

Lemma 5.3: B(r,t) is the graph of an r^t-transposition protocol.

Proof (sketch): For $t = 1$ the result is obvious. For $t \geq 2$ assume the claim inductively for $t - 1$. Let the r^t tokens initially at input i be numbered from 0 to $r^t - 1$, according to their destinations. Input i sends tokens jr^{t-1}, $jr^{t-1}+1,\ldots,(j+1)r^{t-1}-1$ to the j^{th} copy of B(r,t-1), for $0 \leq j < r$. By the inductive hypothesis the copies of B(r,t-1) are r^{t-1}-transposition networks, but with vertex capacity r^t.

Theorem 5.2: Let n and k be such that $r = n^{1/k}$ is an integer. Then there exists and n-transposition network of depth k and $kn^{1+1/k}$ edges.

Proof: The proof is immediate from lemmas 5.2 and 5.3.

Finally, we note that if vertices at each level of this construction represent the same physical set of registers then kn edges may indeed be deleted from $B(r,k)$, and the lower and upper bounds are again identical.

5. Nonconservative Networks

For the nonconservative case we view the registers as black boxes which take inputs of a fixed length n and produce an output of at most n bits. At each step in the computation a register executes

Receive all packets of bits sent at the previous step;
Compute any function of the inputs, producing an output of length $\leq n$;
Pack the outputs into new packets, one for each intended receiver;
Send all new packets.

There is a one-one *physical* correspondence between outputs at a given level in the computation graph and inputs at the subsequent level.

We have obtained results for nonconservative networks only with the severe restriction that each level contain at most n vertices.

5.1. Networks of Depth 2

Let N be a nonconservative n-matrix transposition network of depth 2. We let $\mathbf{X}$ denote the inputs to level 0 (and therefore to the entire network), $\mathbf{Y}$ denote the outputs of level 1, and $\mathbf{Z}$ denote the outputs of level 2 (i.e., the outputs of N). We formalize these definitions as follows.

Notation: Let R_{ki} denote the i^{th} register at level k, $1 \leq i \leq n$, $k = 0,1,2$. Let $\alpha(k,i,j)$ (respectively, $\beta(k,i,j)$) be the j^{th} input (respectively, output) of R_{ki}, $0 \leq j < n$. Then

$$\mathbf{A}\ (i,j : 0 \leq i,j < n : x_{ij} \equiv \alpha(0,i,j))$$
$$\mathbf{A}\ (i,j : 0 \leq i,j < n : y_{ij} \equiv \beta(1,i,j))$$
$$\mathbf{A}\ (i,j : 0 \leq i,j < n : z_{ij} \equiv \beta(2,i,j))$$

Lemma 5.4: Each register R_v computes a function f_v which maps strings in $\{0,1\}^n$ to strings in $\{0,1\}^n$ in a one to one and onto fashion.

Proof: At every level there are at most n registers, each with a capacity of n bits. The outputs ($\mathbf{Z}$) must be computable from the information at each level, and the inputs ($\mathbf{X}$) are computable from the outputs. Thus there must be exactly n registers at each level and the information in a register is incompressible.

A simple extension of this argument yields the following corollary.

Corollary 5.1: For all $\gamma, \delta \in \{\alpha, \beta\}$ and for all g, h, $0 \leq g, h \leq 2$, there exists a 1-1 onto mapping of configurations of $\bigcup_{i,j} \gamma(g,i,j)$ to $\bigcup_{i,j} \delta(h,i,j)$.

In the following, C is always a configuration of $\mathbf{Y}$, D is the corresponding configuration of $\mathbf{X}$, and D^T is the transpose of the matrix represented by D. Then D^T is the configuration of $\mathbf{Z}$ corresponding to C. For every subset $Y \subset \mathbf{Y}$, let C_Y denote the projection of C onto the elements of Y.

Definition:

$\Gamma(Y) = \{x : x \in \mathbf{X} \text{ and } \mathbf{E}\ (D_0, D_1 :: (D_0)_{\{x\}} = 0 \text{ and } (D_1)_{\{x\}} = 1 \text{ and } (C_0)_{\{\mathbf{Y}-Y\}} = (C_1)_{\{\mathbf{Y}-Y\}})\}$,
where the C_i's and D_i's are configurations of $\mathbf{Y}$ and $\mathbf{X}$, respectively.

Lemma 5.5: $\mathbf{A}\ Y \subseteq \mathbf{Y} : |\Gamma(Y)| \geq |Y|$.

Proof: Let $k = |Y|$. There are 2^k assignments to $\mathbf{X}$ consistent with $C_{\{\mathbf{Y}-Y\}}$, so the values of at least k variables cannot be determined from these $n - k$ bits alone.

Corollary 5.2: There exists a matching $m: \mathbf{Y} \to \mathbf{X}$ such that $m(y) \in \Gamma(\{y\})$.

Proof: Immediate from lemma 5.5 and Hall's theorem ([15]).

Corollary 5.3: There exists a matching $r: \mathbf{Y} \to \mathbf{Z}$ such that if $r(y) = z_{ji}$, then the value of z_{ji} cannot be determined from $C_{\mathbf{Y}-\{y\}}$.

Proof: As the values of x_{ij} and z_{ji} are identical, r is easily obtained from the matching given by corollary 5.2: $r(y) = z_{ji}$ iff $m(y) = x_{ij}$.

Lemma 5.6: If $r(y_{ab}) = z_{ji}$ then there is an edge from R_{1a} to R_{2j}.

Proof: By definition of r there exists a configuration of $\mathbf{Y}$ for which register R_{1a} contains information essential to the computation of z_{ji}.

Lemma 5.7: If $m(y_{ab}) = x_{ij}$, then there is an edge from R_{0i} to R_{1a}.

Proof: Let N^{-1} denote the inverse of N, obtained by reversing the direction of all edges in N and inverting the function computed by each register. N^{-1} is an n-transposition network. Since for some configuration of $\mathbf{Y}$ there is information in R_{1a} without which the value of x_{ij} cannot be determined, there must be a path from R_{1a} to R_{0i} in N^{-1}.

Together, m and r yield a set of paths from x_{ij} to z_{ji}, for all $0 \leq i,j < n$. Let $\mu(i,j,R_v) = 1$ iff R_v is on the path from x_{ij} to z_{ji} given by m and r. By lemmas 5.6 and 5.7,

$$m(y_{ab}) = x_{ij} \Rightarrow r(y_{ab}) = z_{ji}$$
$$\Rightarrow \mu(i,j,R_{0i}) = \mu(i,j,R_{1a}) = \mu(i,j,R_{2j}) = 1,$$

and for any register R_v,

$$\sum_{0 \leq i,j < n} \mu(i,j,R_v) = n.$$

The existence of these "physical" paths, together with the bound on $\sum_{i,j} \mu(i,j,R_v)$, reduces the problem to the conservative case.

5.2. Extension to Networks of Depth 3

Let N be a nonconservative n-matrix transposition network of depth 3. Let $\mathbf{X}$ and $\mathbf{Y}$ be as before, and let $\mathbf{Z}$ again denote the outputs of the network, this time at level 3. Let $\mathbf{W}$ denote the inputs to level 2. Formally,

$$\mathbf{A}\ (i,j : 0 \le i,j < n : w_{ij} \equiv \alpha(2,i,j))$$
$$\mathbf{A}\ (i,j : 0 \le i,j < n : z_{ij} \equiv \beta(3,i,j)).$$

Let m again define a matching from $\bigcup_{i,j} \beta(1,i,j)$ to $\bigcup_{i,j} \alpha(0,i,j)$ (i.e. from $\mathbf{Y}$ to $\mathbf{X}$), and let s be defined by

$$s(w_{cd}) = z_{ji} \text{ iff } w_{cd} \equiv y_{ab} \text{ and } m(y_{ab}) = x_{ij},$$

where $w_{cd} \equiv y_{ab}$ means that w_{cd} is the same physical bit as y_{ab} (y_{ab} was sent from R_{1a} to R_{2c} at the end of step 1). Again, the matchings r and s, together with the $\equiv$ relation, yield a *physical* path from input x_{ij} to output z_{ji}. The definition of $\mu(i,j,R_v)$ can be extended accordingly and the problem reduced to the conservative case for $k = 3$.

Theorem 5.3: Any n-transposition network of depth $k \le 3$ requires at least $kn^{1+1/k}$ edges.

Proof: For $k = 1$ the proof is immediate, since there must be a path from R_{0i} to R_{1j} for all $0 \le i,j < n$. The problem has been shown to reduce to the conservative case for $k = 2$ and $k = 3$.

The technique of reducing the nonconservative case to the conservative case cannot immediately be extended to networks of depth greater than 3. The difficulty lies in finding a nice relationship between a matching from $\mathbf{X}$ to $\bigcup_{i,j} \beta(i,j,k)$ and a matching from $\mathbf{X}$ to $\bigcup_{i,j} \beta(i,j,k+1)$, for arbitrary k. We were able to succeed in the depth 2 and 3 cases only because none of the registers are fully general: they are all either network terminals or adjacent to terminals.

References

[1] Ben-Or, M. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. Proc. Second ACM Symp. on the Principles of Distributed Computing (1983), pp. 27-30.

[2] Bracha, G., and Toueg, S. Resilient Consensus Protocols. Proc. Second ACM Symp. on the Principles of Distributed Computing (1983), pp. 12-26.

[3] Cook, S., and Dwork, C. Bounds on the Time for Parallel RAM's to Compute Simple Functions. Proc. Fourteenth Annual ACM Symposium on the Theory of Computing (1982), pp. 231-233.

[4] Cook, S., Dwork, C., and Reischuk, R. Upper and Lower Time Bounds for Parallel RAM's Without Simultaneous Writes. *Submitted for publication.*

[5] Dolev, D. The Byzantine Generals Strike Again. *Journal of Algorithms 3* 1 (1982), pp. 14-30

[6] Dolev, D., Dwork C, and Stockmeyer, L. On the Minimal Synchronism Needed for Distributed Consensus. Proc. 24th IEEE Symposium on the Foundations of Computer Science (1983), pp. 393-402 (revised 10/83).

[7] Dolev, D., Fischer, M., Fowler, R., Lynch, N., and Strong, R. Efficient Algorithms for multiple Processor Agreement. IBM Res. Rep. RJ3428, 1982.

[8] Dolev, D., and Reischuk, R. Bounds on Information Exchange for Byzantine Agreement, Proc. First ACM Symposium on the Principles of Distributed Computing (1982), pp. 132-140.

[9] Dolev, Reischuk, R., and Strong, H.R. Eventual is Earlier Than Immediate. Proc. Twenty-third IEEE Symposium on the Foundations of Computer Science (1982), pp. 196-302.

[10] Dolev, D. and Strong, H.R. Distributed Commit with Bounded Waiting. Proc. Second IEEE Symposium on Reliability in Distributed Software and Database Systems (1982), pp. 53-60.

[10a] Dolev, D. and Strong, H.R. Polynomial Algorithms for Multiple Processor Agreement. Proc. Fourteenth Annual ACM Symposium on the Theory of Computing (1982), pp. 401-407.

[11] Dwork, C., and Skeen, D. The Inherent Cost of Nonblocking Commitment. Proc. Second ACM Symposium on the Principles of Distributed Computing (1983), pp. 1-11.

[12] Dwork, C., and Skeen, D. The Complexity of Asynchronous Consensus. *In preparation.*

[13] Fischer, M., Lynch, N., and Paterson, M. Impossibility of Distributed Consensus with One Faulty Process. Proc. Second ACM Symposium on the Principles of Database Systems (1983), pp. 1-7.

[14] Fortune, S., and Wylie, J. Parallelism in Random Access Machines. Proc. Tenth ACM Symposium on the Theory of Computing (1978), pp. 114-118.

[15] Hall, P. On Representatives of Subsets. *J. London Math. Soc.*, 10 (1935), pp. 26-30.

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

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

1981.

[18] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems 4* 3 (1982), pp. 382-401.

[19] Lynch, N., Fischer, M., and Fowler, R., A Simple and Efficient Byzantine Generals Algorithm. Proc. Second IEEE Symposium on Reliability in Distributed Software and Database Systems (1982), pp. 46-52.

[20] Pease, M., Shostak, R., and Lamport, L. Reaching Agreement in the Presence of Faults. *JACM 27* 2 (1980), pp. 228-234.

[21] Pippenger, N., and Yao, A. Rearrangeable Networks with Limited Depth. IBM Research Report RJ3169, 1981.

[22] Rabin, M. Randomized Byzantine Generals. Proc. 24th IEEE Symposium on the Foundations of Computer Science, (1983), pp. 403-409.

[23] Reischuk, R. A Lower Time-Bound for Parallel Random Access Machines without Simultaneous Writes. IBM Research Report, San José, 1982.

[24] Shrivastava, S.K. A Dependency, Commitment, and Recovery Model for Atomic Actions. Proc. Second IEEE Symposium on Reliability in Distributed Software and Database Systems (1982), pp. 112-119.

[25] Skeen, D. A Decentralized Termination Protocol. Proc. First IEEE Symposium on Reliability in Distributed Software and Database Systems (1981), pp. 27-32.

[26] Skeen, D. 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.

[27] Stockmeyer, L. *Private communication*, August 14, 1983.

APPENDIX A

Existence of a Bivalent Initial Configuration

Initialization Lemma: P has a bivalent initial configuration.

Proof:

"Assume not. Then P must have both 0-valent and 1-valent initial configurations by the assumed partial correctness. Let us call two initial configurations *adjacent* if they differ only in the initial value x_p of a single process p Any two initial configurations are joined by a chain of initial configurations, each adjacent to the next. Hence, there must exist a 0-valent initial configuration C_0 adjacent to a 1-valent configuration C_1. Let p be the process in whose initial value they differ.

"Now consider some admissible deciding run from C_0 in which process p takes no steps, and let σ be the associated schedule. Then σ can be applied to C_1 also, and corresponding configurations in the two runs are identical except for the internal state of process p. It is easily shown that both runs eventually reach the same decision value. If the value is 1, then C_0 is bivalent; otherwise, C_1 is bivalent. Either case contradicts the assumed nonexistence of a bivalent initial configuration."

Note: The statement of the lemma and its proof are taken directly from [13].

APPENDIX B

Construction of Admissible Non-Deciding Run

"The run is constructed in stages, starting from an initial configuration. We ensure that the run is admissible in the following way. A queue of processes is maintained, initially in an arbitrary order, and the message buffer in a configuration is ordered according to the time the messages were sent, earliest first. Each stage consists of one or more process steps. The stage ends with the first process in the process queue taking a step in which, if its message queue was not empty at the start of the stage, its earliest message is received. This process is then moved to the back of the process queue. In any infinite sequence of such stages every process takes infinitely many steps and receives every message sent to it. The run is therefore admissible. Our problem of course is to do this in such a way as to avoid a decision ever being reached.

"Let C_0 be a bivalent initial configuration whose existence is assured by [the initialization lemma]. Execution begins in C_0, and we ensure that every stage begins from a bivalent configuration. Suppose then that configuration C is bivalent and that process p heads the priority queue. Let m be the earliest message to p in C's message buffer, if any, and $\emptyset$ otherwise. Let $e = (p, m)$. By lemma 3, there is a bivalent configuration C' reachable from C by a schedule in which e is the last event applied. The corresponding sequence of steps defines the stage.

"Since each stage ends in a bivalent configuration, every stage in the construction of the infinite schedule succeeds. The resulting run is admissible, and no decision is ever reached."

Note: The construction is taken directly from [13].

APPENDIX C

Notation for the Algorithms

Statement: skip
Semantics: empty

Statement: $x := E$
Semantics: Assignment.

Statement: $x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n$
Semantics: Multiple assignment: $\mathbf{A}(i : 1 \leq i \leq n : x_i := e_i)$.

Statement: **if** B **then** SL **fi**
Semantics: Conditional statement. If the guard B is true, then the statement list SL is executed, otherwise not.

Statement: **if** $B_1 \rightarrow SL_1$ [] $B_2 \rightarrow SL_2$ [] ... [] $B_n \rightarrow SL_n$ **fi**
Semantics: Conditional statement. All n guards are evaluated. If exactly one is true, then the corresponding statement list is executed. If more than one guard is true, then nondeterminism is introduced. All algorithms in the thesis are deterministic, and the guards are always defined.

Statement: **do** $B \rightarrow SL$ **od**
Semantics: Iteration. If the guard is true initially then the statement list is executed repeatedly until guard becomes false, if ever. If the guard is initially false then the statement list is never executed.

Assertions are informal. Assertions and comments are enclosed by brackets ({,}).

than any other processor. Then every processor q, $q \neq z$, must either write to z or receive explicit confirmation that the message has been sent to z. Implicit confirmation is clearly insufficient: z cannot send an implicit message if it does not know there is a broadcast protocol in progress, while an implicit message from any processor other than z is meaningless if the sender fails. By choice of z this confirmation can not be obtained from the first message received by q, since z was not informed before q. Thus q must receive at least two messages or receive one and write to z.

Let $G = (V,E)$ be a directed multigraph with n vertices, one for each processor and named accordingly. G has a directed edge (x,y) for every message sent by processor x to processor y. For all processors w distinct from z, vertex w either has indegree at least 2 or has indegree 1 and $(w,z) \in E$. Thus $|E| \geq 2(n-1)$. Finally, we subtract one message, as, contrary to our assumption, the source processor does not initially receive a message. The total number of messages is therefore at least $2n - 3$, as was to be shown.

Theorem 4.6: Reliable broadcast requires at most $2n-3$ messages in the absence of failures.

Proof: Figure 4.5 shows a failure-free execution of a reliable broadcast protocol requiring $2(n-1)$ messages. The broadcast originates at p_0 at time 0 and the message is sent to p_i, $1 \leq i < n$, at time $i-1$. p_i receives the message at time i, and commits at that time. If p_i,

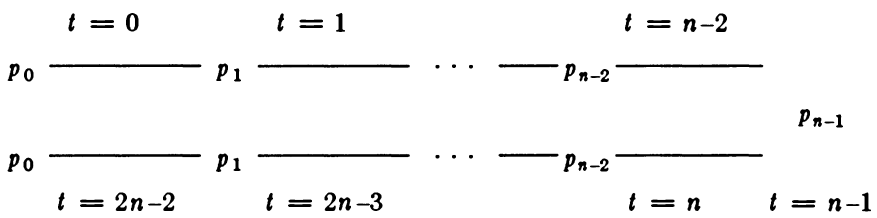
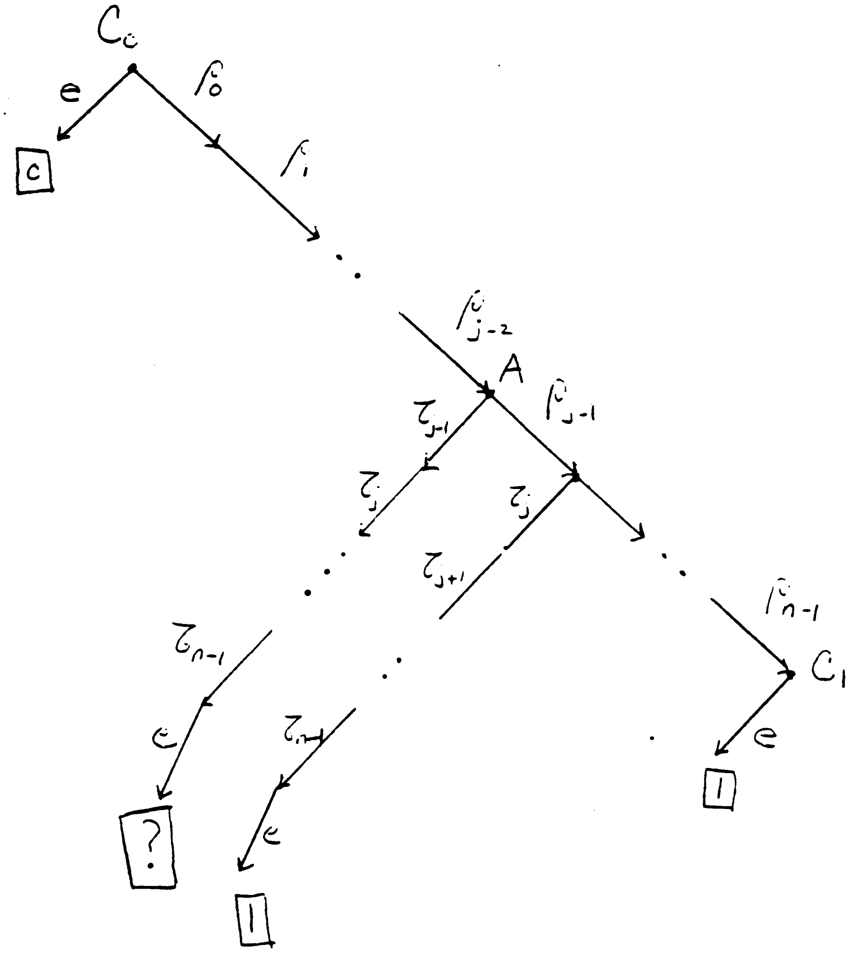


Figure 4.5: Failure-Free Execution of a Reliable Broadcast Protocol



C0
e
ρ0
c
ρ1
ρj-2
A
ρj-1
τj-1
τj
τj
τj+1
τn-1
e
?
τn-1
e
1
ρn-1
C1
e
1