A Simple and Efficient Randomized Byzantine Agreement Algorithm

BENNY CHOR AND BRIAN A. COAN

Abstract—A new randomized Byzantine agreement algorithm is presented. This algorithm operates in a synchronous system of n processors, at most t of which can fail. The algorithm reaches agreement in $O(t/\log n)$ expected rounds and $O(n^2t/\log n)$ expected message bits independent of the distribution of processor failures. This performance is further improved to a constant expected number of rounds and $O(n^2)$ message bits if the distribution of processor failures is assumed to be uniform. In either event, the algorithm improves on the known lower bound on rounds for deterministic algorithms. Some other advantages of the algorithm are that it requires no cryptographic techniques, that the amount of local computation is small, and that the expected number of random bits used per processor is only one. It is argued that in many practical applications of Byzantine agreement, the randomized algorithm of this paper achieves superior performance.

Index Terms—Byzantine agreement, fault-tolerance, randomized algorithms.

I. Introduction

BYZANTINE agreement is the problem of reaching a consensus in a distributed system of n processors, at most t of which may fail in arbitrary, even malicious, ways. In a randomized Byzantine agreement algorithm each processor can use the results of local random coin tosses in the course of reaching agreement. In this paper we present a synchronous randomized Byzantine agreement algorithm that terminates in an expected $O(t/\log n)$ rounds and that works for any $n \geq 3t + 1$. This contrasts with the lower bound of $t + 1$ rounds for deterministic algorithms that was shown by Fischer and Lynch [8]. Our algorithm is of significance because it is simple and efficient enough to be of practical use, because it performs better than any possible deterministic algorithm, because it operates for practical values of n and t, and because it does not require any cryptographic techniques.

Ben-Or [2], Bracha and Toueg [5], and Rabin [12] each have investigated the randomized Byzantine agreement problem. Their primary interest was asynchronous algorithms, but simple variants of their algorithms operate in the synchronous case. Because the topic of this paper is synchronous algorithms, we will discuss only the synchronous variants of their algorithms. A full comparison of the algorithms appears in Section XI; however, we give some advantages of our algorithm here. Our algorithm improves on Ben-Or's in that it operates efficiently for practical ratios of n and t, and it improves on Rabin's in that it does not require initial distribution of coin tosses by a trusted failure-free dealer. In the synchronous case, the algorithm due to Bracha and Toueg is similar to the one due to Ben-Or. For simplicity, we will only make comparisons to Ben-Or's algorithm.

Bracha [4] has recently devised a new synchronous randomized Byzantine agreement algorithm that terminates in an expected $O(\log n)$ rounds. His algorithm assumes a model of computation in which cryptographic techniques are used to conceal information from malicious faulty processors. This is different from the model that we assume in this paper.

In evaluating a Byzantine agreement algorithm, it is often useful to consider the total number of processors that is required by the system in order to tolerate t processor failures. This motivates us to define the *redundancy* of a system of n processors as $(n - 1)/t$. Lamport, Shostak, and Pease [9] have shown that no deterministic nonauthenticated algorithm is possible unless the redundancy is at least 3. An easy extension of their proof shows that this same amount of redundancy is required for randomized algorithms. It is usually desirable to minimize the redundancy in a system in order to reduce the cost of hardware. It is always possible to increase the redundancy in a system and maintain correct operation of a Byzantine agreement algorithm. The basic algorithm that we will present operates for any redundancy of 3 or more. We also present a variant which requires redundancy 6 but terminates twice as fast as the basic algorithm.

If we postulate the existence of a global reliable random coin toss, Byzantine agreement can be reached in a constant expected number of rounds using techniques developed by Ben-Or and by Rabin. Unfortunately, such a source of random coin tosses is unlikely to be available in a distributed system subject to Byzantine faults. A central component of the known randomized Byzantine agreement algorithms is the simulation of a global reliable coin toss using other methods. Ben-Or and Rabin each have techniques for producing global random coin tosses.

Ben-Or uses a technique which works well when the redundancy is high but which has a low probability of producing a good toss when the redundancy is low. In this technique, each processor tosses a coin independently. If one result predominates by a sufficient margin then a coin toss has been produced; otherwise, no usable coin toss has been produced. A spread of sufficient size is likely only when the total number

Manuscript received May 19, 1984; revised February 25, 1985. This research was supported by the National Science Foundation under Grants DCR-8302391 and MCS-8006938, by an IBM graduate fellowship, by the U.S. Army Research Office under Contract DAAG29-84-K-0058, by the Office of Naval Research under Contract N00014-85-K-0168, and by the Advanced Research Projects Agency of the Department of Defense under Contract N00014-83-K-0125.

The authors are with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

0098/5589/85/0600-0531$01.00

of processors is large relative to the number of faults. As a consequence, his algorithm either requires a large (exponential) number of rounds or a high amount of redundancy (equal to $\sqrt{n}$).

Rabin produces global coin tosses efficiently; however, he assumes a different, more powerful model of computation. His coin tosses are precomputed by a trusted dealer who splits the results of each coin toss so that $t + 1$ processors can determine the result, but t processors have no information. The distributed split coin tosses are an expensive resource that is partially consumed at each execution of the algorithm. In some applications, it may be unrealistic to assume that a trusted dealer exists. Rabin's algorithm is a good choice in those applications where his model seems realistic. The algorithm which we will describe generates global coin tosses more efficiently than Ben-Or's algorithm and uses these global coin tosses in a way similar to Ben-Or's method. Our algorithm requires none of the extra machinery of Rabin's algorithm. In particular, no reliable trusted dealer is required. One way in which our technique is inferior to theirs is that ours works only in a synchronous system while theirs work in both synchronous and asynchronous systems. There is something inherent in our technique that does not seem to generalize to the asynchronous case.

A contribution of this paper is our new technique for generating random coin tosses that (while they are not perfect) are of sufficient quality to permit our algorithm to make rapid progress. We now outline our technique for tossing coins. At each round, a small group of g processors is assigned the task of coin tossing. Each processor in this group tosses its own coin and broadcasts the result. The coin toss generated by this group is the majority of individual outcomes. If more than half the processors are faulty, they can bias the coin toss any way they want or cause some processors to see heads and others to see tails. But, if fewer than half are faulty, there is a sufficiently large probability that all correct coin tossers will have the same outcome (provided g is not too big). If they all happen to have the same outcome, the majority is determined regardless of what the faulty processors do. With no more than t faulty processors, there can not be more than $2t/g$ disjoint groups with a majority of faulty processors. After at most that many tosses, a group of coin tossers with a majority of correct processors will be reached. We will show that this leads to fast termination of the algorithm.

We now give a brief outline of the remainder of the paper. In Section II we define the synchronous randomized Byzantine agreement problem. In Section III we elaborate on our fault model. In Section IV we present the basic algorithm. In Section V we analyze the performance of the basic algorithm. In Section VI we present an alternative algorithm that doubles the speed of the basic algorithm at the cost of requiring twice the amount of redundancy. In Section VII we explain how the performance of our algorithm improves if the actual number of faults is smaller than the bound on the number of faults t. In Section VIII we show that it is possible to achieve coordinated termination (all processors decide in the same round) with high probability. In Section IX we present an alternative analysis of the basic algorithm for the case where processor failures are assumed to be uniformly distributed. In Section X we discuss the problem of reintegrating repaired processors into the algorithm. In Section XI we evaluate the basic algorithm by comparing it to the alternatives.

II. The Problem

A synchronous randomized Byzantine agreement algorithm is run by a distributed system of n processors, at most t of which may fail. The computation proceeds by the sending and receiving of messages. Message exchange takes place in a series of rounds over a network that is fully connected and reliable. At each round, a processor can toss coins as part of its computation. These coin tosses affect message generation and processor state. Correct processors toss fair coins and send messages according to their programs. Failed processors can send arbitrary messages.

Each processor starts the algorithm with an input value v from a fixed set of legal inputs V. The goal is that after sending some messages each processor will produce an answer. There are two requirements on the answer. The *agreement* condition is that all correct processors produce the same answer. The *validity* condition is that if all correct processors start the algorithm with the same value, then this value will be the answer produced by the correct processors.

III. The Adversary Model

In a real system, we may not be sure what failure modes may occur. Our goal is to show that, for a wide range of possible failures, our algorithm works correctly. We do this by imagining a powerful adversary which can select the components of the system that fail and which can control the behavior of the failed components. We achieve our goal by proving that, no matter what the adversary does, our algorithm behaves correctly.

It should be clear that we cannot allow our adversary unlimited capabilities. If we did, the adversary could cause all of the components in the system to fail and no algorithm would be possible. In the remainder of this section we describe the capabilities of our adversary and then we argue that this choice of adversary is reasonable.

The principal capability of our adversary is that it can select which processors fail. We define this as *subverting* a processor. There are three aspects of the subverting of processors on which we will elaborate: the selection of which processors to subvert, the control of subverted processors, and the computational resources available to the adversary in developing its strategy.

During the execution of the Byzantine agreement algorithm, the adversary can dynamically select which processors to subvert (up to the limit of t faulty processors). The selection can be based on the following: the code executed by the various processors, the messages previously sent by any of the processors, and the internal state of the processors. We require, however, that the adversary select the processors that it will subvert in round r at the beginning of round r. Specifically, we do not allow it to decide which processors to subvert in round r based on the results of any of the coin tosses made in round r.

The messages that are sent by failed processors are under the

control of the adversary. The messages that are sent in round r by failed processors can be based on any of the following: the code, the current (round r) and prior state of any of the correct or faulty processors, and the current (round r) and the prior coin tosses of any of the correct or faulty processors. We only prohibit that the messages of the failed processors be based on future (round $r+1$ or later) coin tosses.

We assume that the adversary can use the optimal strategy based on the information currently available. There is no requirement that this strategy be computable or efficiently computable. This is in contrast to the assumption commonly made in analyzing cryptographic protocols that the adversary is limited to a polynomial amount of computation. The only limitation we impose is that the strategy of the adversary not be based on any ability to accurately predict the outcome of future coin tosses.

We now review the four principal limitations that we impose on the adversary and attempt to justify the reasonableness of these limitations. First, we limit the number of processors that the adversary can subvert to be fewer than a third of the total because it is known that there is no noncryptographic Byzantine agreement algorithm that operates correctly if the adversary can subvert a third or more of the processors. Second, we assume that the communication system is reliable. The justification for this assumption is that the adversary can simulate a faulty communications link by subverting one of the two adjacent processors. Third, we assume that the adversary cannot accurately predict the future coin tosses of the correct processors. An adversary that could base its decisions on future random choices could negate the benefit of randomization. The whole advantage of using randomization is that it makes the future execution of the algorithm be unpredictable. Finally, we assume that the adversary may not subvert a processor during a round. This assumption is justified by the synchrony of the system. The adversary does not have time to make the necessary observations and then make its decision. In an asynchronous system, only this last assumption would become unreasonable.

IV. The Basic Algorithm

For simplicity of presentation, the algorithm given here is binary (reaches agreement on one bit). It can easily be extended to be multivalued (reach agreement on arbitrary values) using the technique of Turpin and Coan [15].

First, we give an informal description of the algorithm, then we give the code. The algorithm is organized as a series of epochs of message exchange. Each epoch consists of two rounds. The round structure is provided automatically by the synchronous communications network. In the presentation of the algorithm, epoch and round numbers are shown as the first two components of each message. These should be viewed as implicit. They are shown only to make it easier to discuss the algorithm.

The algorithm is parameterized by g, the group size; n, the number of processors; and t, the number of faults tolerated. It is assumed that $n \geq 3t+1$. The parameter g is used to determine the number of processors in each group of coin tossers. The processors are divided into a maximal number of disjoint groups of g processors each. Any processors that are left over belong to no group. The groups are numbered from 1 to $\lfloor n/g \rfloor$.

In each epoch, the processors cooperate to perform a distributed coin toss. In epoch e, the group whose index is congruent to e modulo $\lfloor n/g \rfloor$ actively performs the coin toss. All processors attempt to observe the result. Each processor in the active group tosses a coin and broadcasts the outcome. A processor calculates the coin toss of the group as the majority of the individual coin tosses it receives from processors in the group. The value that processor P *observes* for a coin toss is defined to be this majority value seen by P. If a group contains a large number of faulty processors, then they can control the outcome of the toss or cause correct processors to observe inconsistent values. We will, however, show that for suitably selected group size g, a large enough number of groups will toss sufficiently random coins for our purposes.

We describe the algorithm for the processor P. (All processors run the same code.) The variable CURRENT holds the value that processor P currently favors as the answer of the Byzantine agreement algorithm. At the start of the algorithm CURRENT is set to the input value of processor P. In the first round of each epoch, processor P broadcasts CURRENT. Based on the round 1 messages received, processor P changes CURRENT. If it sees at least $n-t$ round 1 messages for some particular value, then it assigns that value to CURRENT; otherwise, it assigns the distinguished value "?" to CURRENT. In the second round of each epoch, processor P broadcasts CURRENT and (if required) the result of a coin toss. Based on the round 2 messages received, processor P either changes CURRENT or decides on an answer and exits the algorithm. Let ANS be the most frequent value (other than "?") in round 2 messages received by P. Let NUM be the number of such messages. There are three cases depending on the value of NUM. If NUM $\geq n-t$ then processor P decides on the value ANS and exits the algorithm. If $n-t >$ NUM $\geq t+1$ then processor P assigns the value ANS to the variable CURRENT and continues the algorithm. If $t+1 >$ NUM then processor P assigns the coin toss it received in the current round to the variable CURRENT. The selected coin toss is the one performed by the processor group whose index number if congruent to the current epoch number modulo $\lfloor n/g \rfloor$, the number of processor groups.

The code for processor P with parameters g, n, and t is as follows.

```
1.  procedure BYZANTINE_AGREEMENT(INPUT):
2.  CURRENT ← INPUT
3.  for e ← 1 to ∞ do
4.        broadcast (e, 1, CURRENT)
5.        receive (e, 1, *) messages
6.        if for some v there are ≥n - t messages (e, 1, v)
7.              then CURRENT ← v
8.              else CURRENT ← "?"
9.        if GROUP(P) ≡ e(mod ⌊n/g⌋)
10.             then TOSS ← TOSS_COIN( )
11.             else TOSS ← 0
12.       broadcast (e, 2, CURRENT, TOSS)
13.       receive (e, 2, *, *) messages
```

14. ANS ← the value $v \neq$ "?" such that $(e, 2, v, *)$ messages are most frequent
15. NUM ← number of occurrences of $(e, 2,$ ANS, $*)$ messages
16. **if** NUM $\geq n - t$ **then** decide ANS
17. **elseif** NUM $\geq t + 1$ **then** CURRENT ← ANS
18. **else** CURRENT ← majority toss from processor group x where $x \equiv e \pmod{\lfloor n/g \rfloor}$

We make several remarks about the algorithm. GROUP is a procedure that takes as input a processor identity and returns the processor's group number. TOSS_COIN is a procedure that takes no argument and returns the result of a random coin toss (0 or 1). In message descriptions, "*" is a wild-card character that matches anything. In order to allow correct processors to stop sending after they decide, we adopt the convention that a processor that sends no message is assumed to vote for the value in the last message that it sent. Therefore, a processor that has decided may stop sending messages after the first round in which it broadcasts its decided value. Other invalid or missing messages present no difficulty. A processor that sends such a message is faulty and therefore could have sent anything. The recipient of such a message may correctly replace it by any valid message.

Define *value* as a legal input to the algorithm, either 0 or 1. Specifically, "?" is not a value.

Lemma 1: During each epoch, at most one value is sent in round 2 (step 12) messages by correct processors.

Proof: Assume at least one value is sent in round 2 of epoch e. Say processor P sends value v. P has seen at least $n - t$ messages $(e, 1, v)$. At least $n - 2t$ of them are from correct processors and are therefore reliably sent to all processors in the system. All processors have at least $n - 2t$ messages $(e, 1, v)$ in a total of n messages. This leaves at most $2t < n - t$ messages for some other value. This is not enough to cause a correct processor to send a round 2 message for a value other than v. □

In Theorem 2 we prove that our algorithm never produces a wrong answer and we prove that in each epoch there is at least one coin-toss value that will terminate the algorithm. The analysis of the expected running time of the algorithm follows in Section V.

Theorem 2: The algorithm has the following three properties.

Validity: If value v is distributed as input to all correct processors, then all correct processors decide v in round 2 of epoch 1.

Agreement: Let e be the first epoch in which a correct processor decides. If correct processor P decides v in epoch e then by round 2 of epoch $e + 1$ all correct processors decide v.

Termination: In any epoch e, there is at least one value which (if it is adopted by all processors executing the assignment in step 18) will cause all correct processors to decide by round 2 of epoch $e + 1$.

Proof: We show that the algorithm satisfies the three conditions.

Validity: Assume that value v is distributed as input to all correct processors. All (at least $n - t$) correct processors broadcast v in rounds 1 and 2 of epoch 1. All correct processors assign v to ANS, set NUM to a value at least $n - t$, and therefore decide v in round 2 of epoch 1.

Agreement: For processor P to decide v in epoch e, it must be the case that P has seen at least $n - t$ messages $(e, 2, v, *)$. At least $n - 2t \geq t + 1$ of the messages are from correct processors and are therefore reliably sent to all processors in the system. No correct processor sends round 2 messages for any value other than v (by Lemma 1). Only failed processors send such messages. Because there are at most t failed processors, there are at most t round 2 messages for any value $v' \neq v$. All correct processors assign v to ANS, set NUM to a value at least $t + 1$, and either decide v in the current epoch or begin the following epoch with the variable CURRENT equal to v. All correct processors will therefore decide v by round 2 of epoch $e + 1$.

Termination: By Lemma 1, there is at most one value, say v, that is broadcast by correct processors in round 2 of epoch e. Any correct processor that does not execute the assignment in step 18 must have ANS = v because it is not possible to get more than t votes for any other value. The value v (if adopted by all processors executing the assignment in step 18 of epoch e) will cause all correct processors to decide v by round 2 of epoch $e + 1$. This is because all correct processors will start epoch $e + 1$ with the value v assigned to the variable CURRENT. □

Based on the termination property shown in Theorem 2, we make the following definitions. Define a *good* value in epoch e as one which (if it is adopted by all processors executing the assignment in step 18) will cause all correct processors to decide by round 2 of epoch $e + 1$. Define a *good* coin toss in epoch e as one which distributes in epoch e a good value to all correct processors. Define a *bad* value or coin toss as one which is not good.

We are now in a position to explain why randomization is necessary for the correct operation of our algorithm. The termination part of Theorem 2 guarantees that in any epoch, say e, there will be at least one good value. This value can, however, be determined by the adversary in round 1 of epoch e. If the coin toss were replaced with some predetermined value, then the adversary could always cause the good value to differ from this predetermined value. In our scheme, this strategy is unavailable to the adversary because the coin toss is not performed until round 2 of epoch e—after the identity of the good value has already been fixed.

V. Analysis of the Algorithm

In this section we analyze the computational resources used by the basic algorithm. At the system level, we calculate the expected number of rounds needed to reach agreement and the expected number of message bits sent. At the processor level, we calculate the amount of internal memory, the number of computation steps, and the expected number of random bits used by each processor. In particular, we show that the expected number of rounds to reach agreement is $O(t/\log n)$ and the expected number of random bits used by each processor is bounded above by 1. The analysis is worst case in the sense that it allows faulty processors to behave maliciously and holds for any distribution of them, as long as the redundancy is at least 3.

We define some terminology and notation. Let $g = 2m + 1$ denote the number of processors in each group. (This relation

between g and m holds for the rest of the paper.) Let c_e be the coin toss generated at epoch e. Denote by p_e the probability that the coin toss c_e is good (will cause termination), and let $q_e = 1 - p_e$.

We say that coin toss c_e is *pure* if (among the g processors that toss in epoch e) there are at least $m + 1$ correct processors that all toss the same value. Recall that $m + 1$ is a majority of a group of coin tossers. The significance of this definition is that the outcome of a pure coin toss is completely beyond the control of the adversary. It is possible that our algorithm may terminate as the result of a toss that is not pure; however, it is within the power of the adversary to prevent this. Therefore, we assume for the purpose of analysis that progress is made only on pure coin tosses. We remark that the outcomes of pure coin tosses are independent.

If there are at most m faulty processors among the coin tossers of epoch e (a majority of the coin tossers are correct), then

$$\frac{1}{2^m} \leqslant Pr(c_e \text{ pure}).$$

By Theorem 2 and the definition of good coin toss

$$Pr(c_e \text{ good} \mid c_e \text{ pure}) = \tfrac{1}{2}.$$

So, the conditional probability that c_e is good given that at most m coin tossers are faulty, is at least $1/2^{m+1}$.

Theorem 3: For group size $g = \log n$, the expected number of rounds to reach agreement, r, is bounded by

$$r < \frac{4t}{\log n} + 4\sqrt{n} + o(1) = O\left(\frac{t}{\log n}\right).$$

Proof: By the definition of a good coin toss, agreement is reached by all correct processors at most one epoch after a good coin toss is achieved. Every epoch consists of two rounds. Therefore, if *Exp* denotes the expected number of epochs until a good coin toss is achieved, then

$$r = 2 \cdot Exp + 2.$$

The expected number of epochs to get a good coin toss is

$$Exp = \sum_{e=1}^{\infty} e \cdot Pr(c_e \text{ is the first good coin toss}).$$

Since pure coin tosses are independent and since $q_k = 1$ for nonpure coin tosses, we get

$$Pr(c_e \text{ is the first good coin toss}) = q_1 q_2 \cdots q_{e-1}(1 - q_e)$$

so

$$\begin{aligned} Exp &= \sum_{e=1}^{\infty} e \cdot q_1 q_2 \cdots q_{e-1}(1 - q_e) \\ &= (1 - q_1) + \sum_{e=2}^{\infty} (e \cdot q_1 q_2 \cdots q_{e-1} - e \cdot q_1 q_2 \cdots q_e) \\ &= 1 + \sum_{e=1}^{\infty} q_1 q_2 \cdots q_e. \end{aligned}$$

For any specific e, an adversary can *block* the coin toss generated at epoch e by assigning $m + 1$ faulty coin tossers, thus making $q_e = 1$. However, with a limited supply of faulty processors, such blocking can not be repeated indefinitely.

We start by analyzing the first $\lfloor n/g \rfloor$ coin tosses, which are performed by disjoint groups of processors. To do this we calculate an upper bound on $\sum_{e=1}^{\lfloor n/g \rfloor} q_1 q_2 \cdots q_e$. If $q_i < q_{i+1}$ for some i, then the sum can be strictly increased by exchanging q_i and q_{i+1}. Therefore, we assume that the sequence of q_i's is nonincreasing. There can be at most $\lfloor t/(m+1) \rfloor$ groups with a majority of faulty processors. These groups are blocked (the probability of getting a bad coin toss is 1). After the blocked groups, all groups must have at least $m + 1$ good processors and therefore at most $1 - 1/2^{m+1}$ probability of producing a bad toss. We permit the adversary an infinite number of such groups and calculate

$$\sum_{e=1}^{\lfloor n/g \rfloor} q_1 q_2 \cdots q_e \leqslant \frac{t}{m+1} + 2^{m+1}.$$

After $\lfloor n/g \rfloor$ groups, we cycle back through the coin tossers, and so

$$\begin{aligned} &\sum_{e=1}^{\infty} q_1 q_2 \cdots q_e \\ &\quad = \left(\sum_{e=1}^{\lfloor n/g \rfloor} q_1 q_2 \cdots q_e\right) \\ &\qquad \cdot (1 + q_1 q_2 \cdots q_{\lfloor n/g \rfloor} + (q_1 q_2 \cdots q_{\lfloor n/g \rfloor})^2 + \cdots) \\ &\quad \leqslant \left(\frac{t}{m+1} + 2^{m+1}\right)\left(1 + \sum_{i=1}^{\infty} (q_1 q_2 \cdots q_{\lfloor n/g \rfloor})^i\right). \end{aligned}$$

With t faulty processors, at most $\lfloor t/(m+1) \rfloor$ groups of coin tossers can be blocked. Therefore

$$\begin{aligned} q_1 q_2 \cdots q_{\lfloor n/g \rfloor} &\leqslant \left(1 - \frac{1}{2^{m+1}}\right)^{\lfloor n/g \rfloor - \lfloor t/(m+1) \rfloor} \\ &< \left(1 - \frac{1}{2^{m+1}}\right)^{(n-2t)/g}. \end{aligned}$$

Using the bound $t < n/3$, we have

$$q_1 q_2 \cdots q_{\lfloor n/g \rfloor} < \left(1 - \frac{1}{2^{m+1}}\right)^{n/3g}.$$

Because $g = \log n$ and $m < \frac{1}{2} \log n$, we get ($e = 2.718+$ in the next two equations)

$$\left(1 - \frac{1}{2^{m+1}}\right)^{n/3g} < \left(1 - \frac{1}{\sqrt{n}}\right)^{n/3 \log n} < \left(\frac{1}{e}\right)^{\sqrt{n}/3 \log n}.$$

The last bound implies

$$\sum_{i=1}^{\infty} (q_1 q_2 \cdots q_{\lfloor n/g \rfloor})^i < 2 \cdot \left(\frac{1}{e}\right)^{\sqrt{n}/3 \log n}$$

and therefore

$$\left(\frac{t}{m+1} + 2^{m+1}\right) \sum_{i=1}^{\infty} (q_1 q_2 \cdots q_{\lfloor n/g \rfloor})^i = o(1).$$

TABLE I
COIN TOSSING EFFICIENCY

Expected Number of Tries to Produce a Good Coin Toss							
Processors	Faults	Group Size	E(Tosses)	Processors	Faults	Group Size	E(Tosses)
4	1	1	3.2	55	18	5	8.4
7	2	3	4.0	58	19	7	8.2
10	3	3	4.4	61	20	5	9.0
13	4	3	4.7	64	21	7	8.3
16	5	3	5.1	67	22	7	8.9
19	6	3	5.4	70	23	7	8.6
22	7	3	5.9	73	24	7	9.0
25	8	5	5.7	76	25	7	9.5
28	9	5	6.6	79	26	7	9.3
31	10	5	6.3	82	27	7	9.7
34	11	5	7.1	85	28	7	9.7
37	12	5	6.8	88	29	7	10.0
40	13	5	6.9	91	30	7	10.0
43	14	5	7.5	94	31	7	10.4
46	15	5	7.5	97	32	7	10.7
49	16	7	7.5	100	33	9	10.3
52	17	5	8.1	103	34	9	10.9

Combining all these bounds together, we get

$$Exp < \frac{2t}{\log n} + 2\sqrt{n} + o(1) = O\left(\frac{t}{\log n}\right).$$

The expected number of rounds is therefore bounded above by $4t/\log n + 4\sqrt{n} + o(1)$. □

We remark that by setting $m = (1 - \epsilon)\log n$, we have

$$\frac{t}{m+1} + 2^{m+1} = \frac{t}{(1-\epsilon)\log n} + 2n^{1-\epsilon}.$$

This means that by taking somewhat larger size groups, the constant 1 in the $O(t/\log n)$ expression for the expected number of epochs can be achieved asymptotically. On the other hand, if the group size is slightly decreased by taking $m = (1 - \epsilon)\log n/2$, we get

$$Exp < \frac{2t}{(1-\epsilon)\log n} + 2\sqrt{n^{1-\epsilon}} + o(1).$$

This will be significant for achieving early stopping in case the actual number of faults is very small (see Section VII for details).

Our analysis is somewhat loose in that the adversary can reuse faulty processors. For explicit values of n, t, and g the exact value of Exp can be found by direct computation. In Table I we list some values of Exp for small practical systems. For each system size considered, the table shows the group size which minimizes the expected number of tosses.

We calculate the expected number of message bits sent by the basic algorithm. Individual messages have a constant size. In each round there are $O(n^2)$ messages sent. By Theorem 3, the expected number of rounds is $O(t/\log n)$. Therefore, over the course of the algorithm, an expected $O(n^2 t/\log n)$ message bits are sent. We remark that the message complexity can be improved to $O(nt^2/\log n)$ using relay processors, a technique due to Dolev and Strong [7].

In randomized algorithms, the number of random bits used by each processor is important. Current physical devices for producing random bits are rather slow. If a large number of random bits are required, then pseudorandom number generators are often used. Plumstead [11] showed that the fast linear congruence generators are not secure. After seeing a few outcomes, an adversary can predict the remaining tosses (thus allowing the faulty processors to block all future coin tosses). Secure pseudorandom number generators, based on cryptographic techniques, are known to exist under certain intractability assumptions (see [3] and [1]). However, they require a lot of computation, so we are better off if we can avoid using them altogether. A surprising result is the number of random bits used by our algorithm. For sufficiently large n, the expected number of epochs is bounded above by

$$Exp < \frac{2t}{\log n} + 2\sqrt{n} + o(1) < \frac{n}{\log n}.$$

There are $n/\log n$ groups altogether. Therefore the expected number of times we cycle through all groups of coin tossers is bounded above by 1 if n is large. At each cycle one random bit is used by each processor. Therefore, the expected number of coins tossed by each processor is bounded above by 1. From Table I we calculate that for smaller values of n the expected number of tosses per processor is bounded above by 2. Slow physical generators are good enough then, and it is not necessary to resort to pseudorandom number generators.

The complexity of the internal computation is that of counting small numbers. The amount of internal space required by our algorithm is small. Only $\log n$ memory bits for counting NUM are required. Reintegration of faulty processors is easy because no long histories need be stored (see Section X).

VI. AN ALTERNATIVE ALGORITHM

Our basic algorithm is resilient to $t < n/3$ faults and uses two rounds per epoch. If the number of faulty processors t is bound by $t < n/6$, then one round per epoch suffices. Thus the expected number of rounds is cut by a factor of 2. The code for this case uses two thresholds to which NUM, the number of supporters for the current majority value, are compared. This two-threshold scheme is an adaptation of one by Rabin [12]. If NUM falls between the two thresholds, the coin toss is used to determine the value of the variable CURRENT in the next round. With the two thresholds further than t apart, one of the two possible outcomes of the coin toss is good. The generation of coin tosses is done in the same way as in our basic algorithm. This makes the analysis of the expected number of coin tosses needed to get a good one identical to the analysis in Section V.

The code for processor P with parameters g, n, and t is as follows.

```
1.  procedure BYZANTINE_AGREEMENT(INPUT):
2.  CURRENT ← INPUT
3.  for e ← 1 to ∞ do
4.      if GROUP(P) ≡ e(mod ⌊n/g⌋)
5.          then TOSS ← TOSS_COIN( )
6.          else TOSS ← 0
7.      broadcast (e, CURRENT, TOSS)
8.      receive (e, *, *) messages
9.      ANS ← the value v such that (e, v, *) messages are
            most frequent
10.     NUM ← number of occurrences of (e, ANS, *)
            messages
11.     if NUM ≥ n - t then decide ANS
12.     elseif NUM ≥ n - 2t then CURRENT ← ANS
13.     elseif NUM < n - 3t then CURRENT ← 0
14.     else
```

15. COIN ← majority toss from processor group x where $x \equiv e \pmod{\lfloor n/g \rfloor}$
16. **if** COIN = 0 **then** CURRENT ← 0
17. **elseif** COIN = 1 **then** CURRENT ← ANS

VII. Early Stopping

Our algorithm is resilient to t faults, but the actual number of faulty processors f might be smaller than the upper bound t. A desirable property of any Byzantine agreement algorithm is that agreement be reached early in this case. Dolev, Reischuk, and Strong [6] have studied early termination for deterministic Byzantine agreement algorithms. From the analysis in Section V, it follows that the expected number of rounds to reach agreement in the presence of f faults is bounded above by $4f/\log n + 4\sqrt{n} + o(1)$. Thus early stopping is automatically achieved. Furthermore, for the range $\sqrt{n}\log n \leqslant f < n/3$, agreement is reached in $O(f/\log n)$ rounds.

We can modify the basic algorithm to get uniform $O(f/\log n)$ expected time for termination for all values of f. To do that, we slightly decrease the size of each group of coin tossers by setting $m = (1 - \epsilon)\log n/2$. As analyzed in Section V, this yields

$$Exp < \frac{2f}{(1-\epsilon)\log n} + 2\sqrt{n^{1-\epsilon}} + o(1).$$

If $f > \sqrt{n}$, then $\sqrt{n^{1-\epsilon}}$ is $o(f/\log n)$ and agreement is reached in $O(f/\log n)$ rounds.

For f in the range $f \leqslant \sqrt{n}$, the standard deviation of $n - f$ unbiased coins is larger than f. If we take $g = n$ (all processors are coin tossers), then with a constant probability (about 1/3) the deviation from the average of the $n - f$ fair coins (tossed by correct processors) is bigger than f. Therefore, there is a constant probability that termination will be reached at the end of each epoch. Thus the expected run time in this case is constant. (This idea is similar to the one used in [2] and [5].)

The situation now is that large values of f can be handled efficiently by using small groups of coin tossers ($g = (1 - \epsilon)\log n$), and small values of f can be handled efficiently by using large groups of coin tossers ($g = n$). However, we want our algorithm to handle both cases, without knowing in advance which one it faces. In order to accommodate both small and large values of f, we will alternate between the small groups of coin tossers and the large ones. In the even epochs, the coin tossers will be taken from the groups of size $(1 - \epsilon)\log n$. In the odd epochs, every processor will be a coin tosser.

To analyze the run time, we distinguish between the two cases. If f is large ($f > \sqrt{n}$), we might as well assume that the odd epochs are useless and contribute nothing to termination. However, the expected time to reach agreement in this case is at most twice the number of even epochs to reach agreement, which is $O(f/\log n)$. Similarly, if $f \leqslant \sqrt{n}$, then the expected number of odd epochs to reach agreement is $O(1)$. By alternating between odd and even epochs, we only increase the hidden constants in these expressions by a factor of 2. Thus, our interleaved algorithm yields the following expression for Exp, the expected number of rounds to reach agreement:

$$Exp \leqslant \begin{cases} O(f/\log n), & \text{if } f > \sqrt{n}; \\ O(1), & \text{otherwise.} \end{cases}$$

VIII. Coordinated Termination

One disadvantage of our algorithm is that even though all processors start the algorithm in exactly the same round, they might be off by one epoch when they terminate. In this section we show that a minor modification of the algorithm yields an almost certain coordinated termination, namely all correct processors halt at exactly the same round with overwhelming probability. This is done without violating the agreement, validity, and termination requirements, which will still be achieved with probability 1. The expected running time is changed only by a small multiplicative constant.

The modification is quite simple. We know that the expected number of epochs to reach agreement in the basic algorithm is at most $2t/\log n + 2\sqrt{n} + o(1)$ and that the tail probability for the number of epochs converges rapidly. For example, the probability that more than $3t/\log n$ epochs will be needed is no greater than $(1/e)^{\sqrt{n}/3\log n}$. In the modified algorithm, each correct processor will just delay the transition to a "halt" state until epoch $3t/\log n$ (in case it made its decision before that epoch) and behave as before otherwise. This guarantees coordinated termination by epoch $3t/\log n$ with probability at least $1 - (1/e)^{\sqrt{n}/3\log n}$. Greater confidence of coordinated termination can be achieved at the cost of more rounds.

It remains to be seen whether Byzantine agreement in $O(t/\log n)$ expected time and probability 1 of coordinated termination can be achieved. The deterministic lower bound implies only $t + 1$ worst case lower bound for any such algorithm.

IX. Uniformly Distributed Processor Failures

In the previous analysis we have assumed that an adversary controls both the selection of which processors fail and the behavior of the failed processors. It does this in the way which will cause our algorithm the most difficulty. An alternative assumption is that the faulty processors are randomly distributed.

In the analysis in this section we assume that the distribution of processor faults is uniform. That is, each of the $\binom{n}{t}$ ways of distributing t faults among the n processors is equally likely. We retain the assumption that failed processors are under the control of the adversary and therefore behave in a way which will cause our algorithm the most difficulty. In Theorem 4 we show that for $g = 1$ and uniform processor faults our algorithm terminates in a constant expected number of rounds. Because $g = 1$, each group of coin tossers consists of a single processor.

Theorem 4: If $g = 1$ and processor faults are uniformly distributed, then the expected number of rounds until the last correct processor decides is at most 8.

Proof: The expected number of rounds is $2c + 2$ where c is the expected number of coin tosses to get a good toss. We show that the expected number of coin tosses is at most 3. The probability that the coin toss of a correct processor is good is at least $\frac{1}{2}$ (by Theorem 2 and the fairness of the coin toss). The probability that processor P is correct is at least $\frac{2}{3}$ because $n \geqslant 3t + 1$. Therefore, the probability that the coin toss of processor P is good is at least $\frac{1}{3}$. The conditional probability that the coin toss of processor P is good given that previous coin tosses have been bad is at least as great as the un-

conditional probability (at least $\frac{1}{3}$). Therefore, the expected number of coin tosses is at most the mean of a geometric random variable with parameter $\frac{1}{3}$, which is 3. □

We calculate the expected number of message bits sent by the algorithm. Individual messages have a constant size. On each round, there are $O(n^2)$ messages sent. By Theorem 4, the expected number of rounds is 8. Therefore, over the course of the algorithm, an expected $O(n^2)$ message bits are sent. This can be reduced to $O(nt)$ using the relay-processor technique of Dolev and Strong [7].

Termination in a constant expected number of rounds is an attractive feature of the algorithm; however, this feature is not unique to randomized algorithms. Under the same assumption of uniform randomly distributed processor failures, a deterministic algorithm due to Reischuk [13] also terminates in a constant expected number of rounds.

X. Reintegration of Failed Processors

It is possible for a processor that fails and is subsequently repaired to rejoin our Byzantine agreement algorithm. We assume that such a processor loses its local memory and that it runs special recovery code after it is repaired. In order to rejoin the algorithm a processor needs to replace its lost state information. This is easy with our algorithm because the amount of state information is small. It is also important that correct processors do not record the identities of known faulty processors. Reintegrating failed processors permits the algorithm to tolerate a larger number of failures as long as at most t occur simultaneously. In this section we describe how a repaired processor rejoins the algorithm.

During an epoch, there are several times at which a repaired processor can rejoin the algorithm. We describe one. The repaired processor simply begins the epoch with step 13, receiving round 2 messages. A processor that fails and is subsequently repaired is counted as a failed processor only from the epoch in which it fails until two epochs after it recovers. After that it is considered to be a correct processor. The constraint is that at any time there are no more than t failed processors.

A processor that attempts to rejoin the algorithm after the correct processors have decided or as they are deciding may not see enough messages in order to decide. To solve this problem, we adopt the rule that a processor that sees $t + 1$ or more silent processors in the epoch after it attempts to rejoin the algorithm will conclude that a decision has been reached. It then broadcasts a query and decides on any value that it receives from at least $t + 1$ processors. It is easy to see that a processor that rejoins the algorithm as a decision is being reached can not be tricked into making a wrong decision. There will not be enough votes for any incorrect value because only faulty processors will cast such votes after a correct processor has decided.

XI. Evaluation

In this section we evaluate the extent to which our algorithm is practical. We do this by comparing our algorithm with four alternatives (all synchronous): Ben-Or's randomized algorithm, Bracha's randomized cryptographic algorithm, Rabin's randomized cryptographic algorithm, and a deterministic algorithm due to Lynch, Fischer, and Fowler.

When the redundancy r of a system of processors is $\Omega(t)$, Ben-Or's algorithm terminates in a constant expected number of rounds. For practical systems, however, it is desirable to operate at a lower redundancy in order to minimize the cost of computer hardware. The optimal value is $r = 3$. Unfortunately, for any r that is $O(1)$, Ben-Or's algorithm requires an exponential number of rounds. Compared to Ben-Or's algorithm, ours is more efficient for practical amounts of system redundancy.

The new randomized algorithm due to Bracha [4] terminates in $O(\log n)$ rounds using cryptographic techniques. The principal advantage of his algorithm is this extremely fast asymptotic performance. The principal disadvantages are the use of cryptographic techniques, a seemingly high constant factor in the run time, relatively high communications costs, and the use of a hard to compute local graph partition. (The only known deterministic algorithms to compute this partition are exponential in run time.)

Rabin's algorithm terminates in a constant expected number of rounds; however, it requires more resources than our algorithm. In particular, it requires a trusted dealer that distributes random coin tosses before the start of the algorithm. The underlying mechanism is authentication and Shamir's [14] shared secret. If this cost seems small, then Rabin's algorithm would be the choice. On the other hand, if the cost seems high, then our algorithm would be the choice. We believe that in practical systems, it is often unrealistic to assume the existence of a trusted dealer.

The most practical deterministic algorithm is due to Lynch, Fischer, and Fowler [10]. There are tradeoffs between their algorithm and ours. The principal advantages of the deterministic algorithm are that it uses a fixed number of rounds and that all processors decide at the same round. The principal advantages of our algorithm are that the expected number of rounds is small, that the expected number of message bits is small, and that only two rounds are required if the input to all processors is the same. Our algorithm cannot ensure synchronous termination; however, using the techniques of Section VIII, this property can be achieved with high probability.

We conclude that for a practical algorithm, one would choose either the deterministic algorithm of Lynch, Fischer, and Fowler or our randomized algorithm. The deterministic algorithm would be chosen if the extra synchronization that it provides were important to the particular application. Otherwise, our randomized algorithm would be chosen. If it seems realistic to assume that processor faults are uniformly distributed, then the randomized algorithm is especially attractive because its expected running time is constant.

Acknowledgment

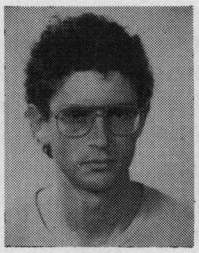
We would like to thank N. Alon, J. Burns, O. Goldreich, L. Levin, B. Lindsay, N. Lynch, M. Tuttle, and W. Weihl for many helpful discussions.

References

[1] W. Alexi, B. Chor, O. Goldreich, and C. Schnorr, "RSA/Rabin bits are $\frac{1}{2} + \frac{1}{poly(\log N)}$ secure," in *Proc. 25th Annu. Symp. Foundations of Comput. Sci.*, Oct. 1984, pp. 449-457.

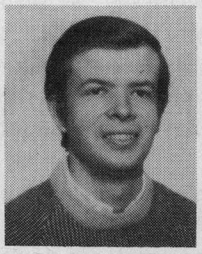
[2] M. Ben-Or, "Another advantage of free choice: Completely asyn-

chronous agreement protocols," in *Proc. 2nd Annu. ACM Symp. Principles of Distributed Comput.*, Aug. 1983, pp. 27-30.
[3] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo-random bits," *SIAM J. Comput.*, vol. 13, pp. 850-864, Nov. 1984.
[4] G. Bracha, "An $O(\log n)$ expected rounds randomized Byzantine generals algorithm," in *Proc. 17th Annu. ACM Symp. Theory of Comput.*, May 1985.
[5] G. Bracha and S. Toueg, "Resilient consensus protocols," in *Proc. 2nd Annu. ACM Symp. Principles of Distributed Comput.*, Aug. 1983, pp. 12-26.
[6] D. Dolev, R. Reischuk, and H. R. Strong, "Eventual is earlier than immediate," in *Proc. 23rd Annu. Symp. Foundations of Comput. Sci.*, Nov. 1982, pp. 196-203.
[7] D. Dolev and H. R. Strong, "Polynomial algorithms for multiple processor agreement," in *Proc. 14th Annu. ACM Symp. Theory of Comput.*, May 1982, pp. 401-407.
[8] M. Fischer and N. Lynch, "A lower bound for the time to assure interactive consistency," *Inform. Processing Lett.*, vol. 14, pp. 183-186, June 1982.
[9] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382-401, July 1982.
[10] N. Lynch, M. Fischer, and R. Fowler, "A simple and efficient Byzantine generals algorithm," in *Proc. 2nd Symp. Reliability in Distributed Software and Database Syst.*, July 1982, pp. 46-52.
[11] J. Plumstead, "Inferring a sequence generated by a linear congruence," in *Proc. 23rd Annu. Symp. Foundations of Comput. Sci.*, Nov. 1982, pp. 153-159.
[12] M. Rabin, "Randomized Byzantine generals," in *Proc. 24th Annu. Symp. Foundations of Comput. Sci.*, Nov. 1983, pp. 403-409.
[13] R. Reischuk, "A new solution for the Byzantine generals problem," IBM Corp., Tech. Rep. IBM-RJ-3673, Sept. 1982.
[14] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, pp. 612-613, Nov. 1979.
[15] R. Turpin and B. Coan, "Extending binary Byzantine agreement to multivalued Byzantine agreement," *Inform. Processing Lett.*, vol. 18, pp. 73-76, Feb. 1984.



Benny Chor received the B.Sc. and M.S. degrees in mathematics from the Hebrew University of Jerusalem, Israel, in 1980 and 1981, respectively.

He is currently working toward the Ph.D. degree in computer science at the Massachusetts Institute of Technology, Cambridge. His research interests include cryptography, complexity theory, and randomized algorithms for distributed systems.



Brian A. Coan received the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1977 and the M.S. degree in computer engineering from Stanford University, Stanford, CA, in 1979.

He has worked for Amdahl Corporation and for AT&T Bell Laboratories. Currently he is working toward the Ph.D. degree in computer science at the Massachusetts Institute of Technology. His research interests are in the theory of distributed computer systems.

Randomized Byzantine Agreement

KENNETH J. PERRY, MEMBER, IEEE

***Abstract*—A randomized model of distributed computation was recently presented by Rabin [8]. This model admits a solution to the Byzantine Agreement Problem for systems of n asynchronous processes where no more than t are faulty. The algorithm described by Rabin produces agreement in an expected number of rounds which is a small constant independent of n and t. Using the same model, we present an algorithm of similar complexity which is able to tolerate a greater proportion of malicious processes. The algorithm is also applicable, with minor changes, to systems of synchronous processes.**

***Index Terms*—Distributed computing, distributed database systems, fault-tolerance, protocols, reliability.**

Manuscript received May 9, 1984; revised February 1, 1985. This work was supported in part by the National Science Foundation under Grant MCS83-03135.

The author was with the Department of Computer Science, Cornell University, Ithaca, NY 14853. He is now with the I. B. M. Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

I. Introduction

CONSIDER a collection of n processes, each possessing a previously initialized local variable *message*. Desired is an *agreement protocol* whose execution results in the same value being assigned to the *message* variables of all processes. Some processes may not obey the protocol for its entire duration. In fact, these processes may actively attempt to hinder agreement. We call any process that deviates from the protocol *malicious*. Those processes that strictly adhere to the protocol for its entire execution are deemed *proper*. Because we can not constrain the behavior of malicious processes, we require only that all proper processes agree on their *message* values upon termination.

A *Byzantine Agreement* protocol is an agreement protocol with the additional constraint that if the protocol commences with the same value V in the *message* variables of all proper

0098-5589/85/0600-0539$01.00