Applications of Byzantine Agreement in Database Systems

HECTOR GARCIA-MOLINA and FRANK PITTELLI
Princeton University
and
SUSAN DAVIDSON
University of Pennsylvania

In this paper we study when and how a Byzantine agreement protocol can be used in general-purpose database management systems. We present an overview of the failure model used for Byzantine agreement, and of the protocol itself. We then present correctness criteria for database processing in this failure environment and discuss strategies for satisfying them. In doing this, we present new failure models for input/output nodes and study ways to distribute input transactions to processing nodes under these models. Finally, we investigate applications of Byzantine agreement protocols in the more common failure environment where processors are assumed to halt after a failure.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; H.2.4 [**Database Management**]: Systems—*distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Byzantine agreement, distributed databases, network protocols

1. INTRODUCTION

Byzantine agreement (BA) is the problem of making a set of processors, some of which may fail in arbitrary ways, agree on a common "value." This problem has recently received considerable attention in the literature (e.g., [6, 7, 12, 17, 23, 25]), mainly because *reliability* has become one of the principal goals of computer systems. A reliable system must be able to perform useful and correct computations in the face of failing components. The agreement problem turns out to be fundamental in reliable computing, and illustrates the subtleties that appear in coping with faulty processors. (Incidentally, the name "Byzantine" refers to a military scenario that was initially used to describe the problem [17].)

The goal of this paper is to study when and how Byzantine agreement protocols can be of use in general-purpose database processing. A number of database (or database-related) applications have been suggested [1, 3, 8, 12, 17–19, 24], but in

Authors' addresses: H. Garcia-Molina and F. Pittelli, Department of Computer Science, Princeton University, Princeton, NJ 08544; S. Davidson, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1986 ACM 0730-0301/86/0300-0027 $00.75

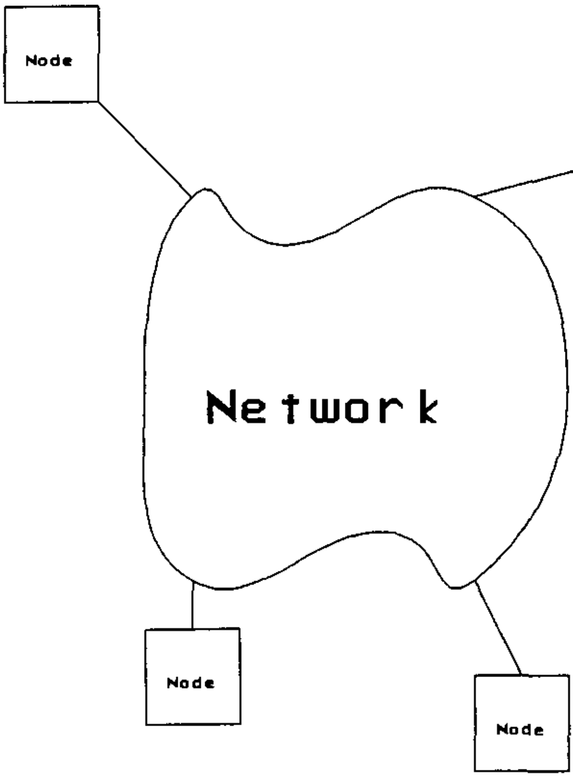

Fig. 1. The system.

many cases these applications have been mentioned in passing, or the practical "details" omitted. Also, there has been considerable controversy in the database community with regards to the applicability of Byzantine agreement, mainly because of the high message overhead of its solutions. Thus we study, from a "practitioner's" point of view, what BA is and what its pragmatic implications are.

The first step in any reliable system design is to define the expected operation of each component during both normal and failure periods. Therefore, in Section 2, we present the *operation models* of our distributed system. Based on these models, we present, in Section 3, one version of the BA problem and outline its solution. (A reader familiar with BA may skim over these two sections.) In Section 4 we look at the database processing application itself and argue that the main application of BA is in the distribution of input transactions to a set of processors with a replicated database. Finally, in Section 5, we look at a failure environment different from that typically used in BA and discuss the uses of BA protocols in such an environment.

2. MODELING FAILURES

Multiple processors are necessary for reliable computing. Therefore, our model of the system consists of a set of processing *nodes* (or computers) connected through a communication *network* (see Figure 1). In this model, all processing needed by the distributed application is performed at the nodes, while any processing needed for communication (e.g., routing) is performed by the network.

2.1 Node Models

Processing nodes can fail in many different and strange ways, and modeling these failures accurately is difficult. One way to avoid these problems is simply to assume that when a node fails it can have arbitrary behavior. In this case, *no assumptions* are made about failures. A failed node can send any message, including garbled or misleading ones, to other nodes. It can refuse to send messages when it is supposed to. It can even collaborate with other failed nodes in an attempt to subvert the entire system.

This model is very simple, and yet very powerful. Unlike other models, it covers any conceivable failure, regardless of whether it was considered by the system designers. It is the most conservative model, and any system that protects against this type of failure will be highly reliable.

We call a node that can fail in this unpredictable way an *insane* node.[1] For the time being we assume that insanity is a permanent property. Later on we consider the case where a node can be repaired and cease to be insane.

Clearly, if all nodes are insane, we are unable to do reliable computing. We need to assume that there are some nodes that can be trusted, even if we do not know which ones they are. A *perfect* node is one that never fails.[2] It always follows the algorithms it is given, and never pauses or halts. Furthermore, the algorithms it follows are correct. A perfect node responds promptly to messages from other nodes (more on this later). Again, we temporarily assume that if a node is perfect, it is perfect for all time.

Lastly, there is a node failure model that lies between perfection and insanity. This model is usually not employed in BA problems; but we discuss its relationship to BA at several points in this paper. A node is considered to be *sane* if it only fails in "clean" well-defined ways.[3] (This type of failure is usually called a *crash.*) When a sane node fails, it simply halts, without ever deviating from its algorithms. It may lose the data contained in memory, but the data contained in *stable storage* [21] are unaffected by the failure. When it is repaired, the node immediately starts executing a predefined recovery procedure. (There are a number of techniques for increasing the probability that a real node behaves like a sane one [27], but, of course, this probability will never be 1.)

2.2 Network Model

In this paper we make the following additional assumptions about the network and timing. We first state the assumptions and then discuss them.

N1. *The network delivers all messages correctly.* Any message sent from node x to y is eventually delivered. Messages are not altered in any way, and the network never generates spontaneous messages.

N2. *Messages are authenticated.* All messages are signed and encoded by senders in such a way that the receiver can determine unequivocally who sent

[1] An insane node is called a *traitor* in [17].
[2] A perfect node is called a *loyal* node in [17].
[3] A sane node is a *fail-stop* node in [27].

the message and what it contained. A third node that simply forwards the message cannot alter it in any way. (A node may, however, refuse to forward a message.)

T1. *Perfect nodes have accurate and synchronized clocks.* Specifically, at any instant the clocks differ by at most τ time units [16].

T2. *The network has a guaranteed delivery time.* The network delivers all messages (between perfect nodes) within T_D time units.

T3. *The processing time of perfect nodes can be bounded.* Given a segment of code s, we can compute the maximum time t_s that a perfect node will take to execute the code.

These assumptions may be "relaxed" by considering any node involved in the violation of an assumption to be insane [12]. For example, if a message is lost or garbled (violating N1), we can say that the sending or receiving node is insane. However, as we see in Section 4, there must always be some perfect nodes, and for these nodes the assumptions must hold [11]. Finally, BA is possible even without assumption N2 [12, 17], but the resulting algorithms are less efficient. Since authentication is practical and well understood, we keep N2.

A number of techniques can be used to increase our confidence in the network assumptions. For example, cryptographic techniques [5] (or error-detecting codes [2], if nodes are not malevolent) can be used to implement message signatures (assumption N2). A signal from a very reliable clock can be periodically broadcast (by radio or a dedicated line to avoid network failures) to synchronize the clocks (assumption T1). As an alternative, a reliable clock synchronization protocol can be used [10, 20, 22].[4] To ensure a maximum message delivery time (assumption T2), the network can be designed with multiple routing paths and spare capacity.

3. BYZANTINE AGREEMENT PROBLEM

In this section we define one version of the BA problem. Our objective is to give the reader an intuitive understanding of the problem and its solution. The reader is referred to the literature for details and proofs of correctness. (The version we present here is from [17].) The practical applications of BA are discussed in the following sections.

The problem is that at time α_0 a *general* node wants to broadcast, in a bounded time, a value v to a set of n *lieutenant* nodes. (The terminology is from the military scenario of [17].) Some of the nodes, including the general node, may be insane. Let m be the maximum number of nodes that are insane. Nodes that are not insane are perfect.

A solution algorithm is considered correct if:

Condition 1. When the algorithm completes, all perfect lieutenants agree on the same value.

Condition 2. If the general is perfect, then all perfect lieutenants agree on the value sent by the general.

[4] Incidentally, these references show that the clock synchronization algorithm does not have to use BA, as had earlier been suspected.

Note that if the general is insane, then the perfect lieutenants can agree on any value, as long as they all agree on that same value. However, if the general is perfect, then the lieutenants must agree on the value broadcast by the general.

The fact that the general may be insane is what complicates solutions to this problem. Instead of broadcasting a single value v, the general may send out a collection of values v_1, v_2, ... v_q to the lieutenants. This implies that the lieutenants cannot simply take the value they receive from the general and consider it correct. Each lieutanant L_j must instead proceed in three steps:

(a) Lieutenant L_j receives a value from the general.
(b) Lieutenants exchange the values they received so that they all (or at least the perfect ones) know the different values $v_1, v_2, \ldots, v_q$ that were broadcast by the general. (As we will shortly see, this step is tricky.)
(c) Once L_j knows the list $v_1, \ldots, v_q$, it applies a fixed rule to obtain the single value it will agree on. (Obviously, all lieutenants will use identical rules.) If the list contains a single value, then L_j must use that value in order to satisfy condition 2. Otherwise, the general is insane because it broadcast different values, and the lieutenants may agree on any value. For instance, they may choose a standard null value, or the average of $v_1, \ldots, v_q$.

Returning to step (b), it is necessary that all perfect lieutenants end up with exactly the same list of values $v_1, \ldots, v_q$, or else they may obtain different values in step (c). The insane lieutenants may interfere with this process in two ways.

The first way is by lying or by masquerading as a different lieutenant. For example, insane lieutenant L_i may tell perfect lieutenant L_j that it received v_1 from the general, and may tell perfect L_k that it received v_2. If we do not take precautions, the lists that L_j and L_k use in step (c) would then be different.

One simple solution to this problem is to use authenticated messages (assumption N2). The general will sign the value it broadcasts in step (a), and when a lieutenant forwards a value (in step (b)), it adds on its signature. The receiver of the forwarded value will then be able to determine the true value sent by the general, as well as the true identity of the forwarding lieutenant. We use the notation $v_1:G:L_i$ for the message containing value v_1, authenticated by the general, G, and by lieutenant L_i.

Authentication by itself does not solve the entire problem. A second way in which an insane lieutenant can cause confusion is by not forwarding values, or by delaying the forwarding. To illustrate, consider the following scenario with three lieutenants (see Figure 2). Suppose that the general and one of the lieutenants, L_1, are insane. At time α_0 the general broadcasts $v_1:G$ to L_1 and $v_2:G$ to L_2 and L_3. Perfect lieutenant L_2 next broadcasts $v_2:G:L_2$ to all other lieutenants. Similarly, L_3 broadcasts $v_2:G:L_3$. However, L_1 sends $v_1:G:L_1$ *only* to L_2. If we take no further precautions, L_2 will finish step (b) with the value list v_1, v_2, while L_3 will only have value v_1.

To avoid this and similar situations, we must introduce two safeguards. The first is to have lieutenants broadcast, not just the value they receive from the general, but also additional values they receive from other lieutenants. In our example, when L_2 receives $v_2:G:L_1$, it will broadcast $v_2:G:L_1:L_2$ to ensure that other lieutenants like L_3 also receive v_2. (Clearly, L_2 does not have to send v_2 to L_1, since it has already seen v_2.)

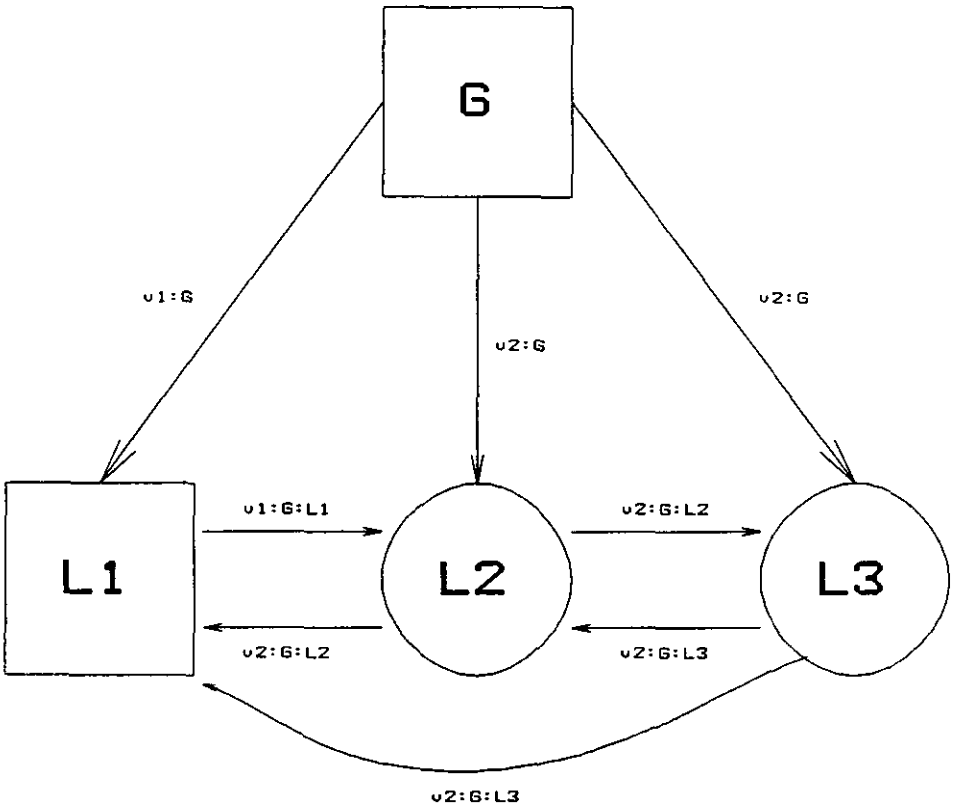

Fig. 2. A BA example.

The second safeguard is to put a limit on the time that lieutenants will wait for new values. Otherwise, lieutenants would never know when they have received all the values for step (c). Recall that α_0 is the start time, t_D the guaranteed delivery time (assumption T2), τ the maximum clock drift (assumption T1), and t_s is the maximum time it takes a perfect node to process and forward a value (assumption T3). Therefore, if a lieutenant L_i has not received a value from the general by time (on its clock) $\alpha_0 + \tau + t_D + t_s$, then L_i knows that the general is insane and can safely ignore any future messages from the general.

Similarly, L_i can ignore messages of the form $v_k{:}G{:}L_j$ received after time $\alpha_0 + \tau + 2(t_D + t_s)$. In this case, L_j, the sender of this message, must be insane and v_k can be ignored. If v_k was sent by L_j to other perfect lieutenants before the time limit, then it will be up to them (and not up to L_j) to send v_k to L_i. In general L_i can ignore any message of the form $v_k{:}G{:}L_1{:}L_2{:}\cdots{:}L_p$ if it arrives after time $\alpha_0 + \tau + (p + 1)(t_D + t_s)$.

When can a lieutenant L_i be certain that it has received *all* necessary values? By the argument above, at time $\alpha_0 + \tau + (m + 1)(t_D + t_s)$, L_i can ignore all messages of the form $v_k{:}G{:}L_1{:}L_2{:}\cdots{:}L_p$, where $p \leq m$ is the maximum number of insane nodes. However, L_i can also ignore messages where $p > m$, for the following reason. If $p > m$, then there are at least two perfect nodes in G,

$L_1, \ldots, L_p$. If the general is perfect, then L_i would have received the value v_k by time $\alpha_0 + \tau + (t_D + t_s)$. If the general is not perfect, then the first of the two perfect lieutenants in $L_1, \ldots, L_p$ would have sent v_k to L_i before $\alpha_0 + \tau + (m + 1)(t_D + t_s)$. (In the worst case, L_m is the first perfect lieutenant. It must have received $v_k{:}G{:}L_1{:}\cdots{:}L_{m-1}$ by $\alpha_0 + \tau + m(t_D + t_s)$ and then sent $v_k{:}G{:}L_1{:}\cdots{:}L_m$ to L_i.) In either case, L_i receives v_k before $\alpha_0 + \tau + (m + 1)(t_D + t_s)$. Therefore, at this time, L_i is certain it has received all values and can proceed to step (c).

In summary, in step (b) lieutenants proceed as follows. Any message $v_k{:}G{:} L_1, \ldots, {:}L_p (p \geq 0)$ that arrives is checked for correctness and timeliness. If the message has the correct format and signatures and if it arrives on time (before $\alpha_0 + \tau + (p + 1)(t_D + t_s)$), then the value in the message is added to the list of values and is broadcast to other lieutenants. Note that $L_1, \ldots, L_p$ do not need to receive the value since they have already seen it. Similarly, if $p \geq m$, the value does not have to be broadcast at all because one of $G, L_1, \ldots, L_p$ is perfect and has already broadcast the value. Finally, at time $\alpha_0 + \tau + (m + 1)(t_D + t_s)$ step (b) completes, and the resulting list of values is passed to step (c) for the final decision.

The algorithm we have outlined guarantees agreement as defined by conditions 1 and 2, even if there are very few perfect nodes. If there is one or no perfect node(s), then conditions 1 and 2 are satisfied trivially. If there are just two perfect nodes, they will reach agreement no matter how many insane nodes there are.

There are a number of other solutions, but they all use similar ideas and have the following two characteristics:

- In all algorithms the worst-case delay for reaching agreement is $\tau + (m + 1)(t_D + t_s)$ time units. However, from a practical point of view, this delay is not critical because in most cases agreement can be reached much sooner. For example, the algorithm we presented can be modified so that, in the common case where there are no failures, a lieutenant can proceed to step (c) as soon as it receives valid messages from the $n - 1$ other lieutenants [9, 12]. (Incidentally, worst-case times are similar in some sane node-recovery algorithms.)
- In all solutions the message traffic is also high, for the value received by a lieutenant from the general must somehow be transmitted to all other lieutenants. This represents the essence of BA: no single node is trustworthy, so all nodes must collect all information and make decisions for themselves. (With sane nodes, on the other hand, a single node can make decisions, thus cutting down on the message traffic.)

In closing this section, we make three observations that will be useful later on:

(i) Perfect nodes cannot agree on the identity of the insane nodes. A given node L_i may establish with certainty that some other node L_j is insane (because it failed to forward a message). However, L_j may appear perfect to other nodes, and L_i has no foolproof way of "convincing" these other nodes that L_j is indeed insane.

(ii) Although it is often overlooked in BA discussions, it is critical that all participants agree beforehand on the start time, α_0, of the agreement algorithm.

If this time is not known, the algorithm can never finish. One practical way to avoid the problem is to have periodic agreements. For example, the BA algorithm can be run starting every second or minute. If the general has no value to broadcast, it can send out a null value. (It could also not send any value at all. This would be acceptable as long as the lieutenants then agreed on a default null value.) A second technique would be to use one BA to agree on both the broadcast value and on the time of the next BA. This is only useful if the general can predict the next time at which it will want to broadcast a value.

In either case, if several executions of the BA algorithm can overlap in time, then the values broadcast by the general must carry an identification that binds them to one execution. The starting time of the execution could be concatenated to the value for this purpose.

(iii) For BA we assume that m or less nodes are insane, and the rest are perfect. It is tempting to consider weaker assumptions, specifically that the nodes that are not insane are sane. Unfortunately, with these weaker assumptions, BA is not possible [11]. Thus, if we wish to cope with m or less insane nodes, we must assume perfect nodes.

4. APPLICATIONS OF BA IN DATA PROCESSING

In the previous section we discussed how nodes could agree on a value. Now we want these nodes to do much more. We want them to do *reliable data processing* in a distributed computing system. By data processing we mean conventional and general-purpose database management [4, 14]. Users submit *transactions* that contain one or more database commands (e.g., print the location of the carrier "Nimitz," withdraw ten million dollars from account 777). Each transaction is run as an atomic unit against the database, and the results are given to the users.

In this section we investigate the applications of BA in this environment. We start by making three simplifying assumptions about transactions.

R1. *Transactions contain user authentication information.* Transactions from unauthorized users are discarded by the database system. Hence we ignore malevolent users in our discussions.

R2. *Each transaction originates from a single user.* The user can be a military commander, a customer at an automatic teller machine, or a company manager.

In some cases a command or transaction (e.g., fire a missile) must come from several users (e.g., the president and a commander) before it can be executed. To satisfy assumption R2, we can simply view this case as a collection of single user transactions. Each transaction records in the database the fact that one more user has authorized the command, and, when the required number is reached, the target command is executed.

R3. *The input/output functions are performed by input/output nodes that are different from the processing nodes.* As we will shortly see, certain computations must be performed when receiving a transaction from a user (input) and when displaying results (output). We consider the nodes that perform these computations to be separate from the data processing nodes, so that we may make different failure assumptions about them. Of course, in reality, a single node may perform all functions.

The next step (as in the previous section) is to specify the types of failures that we consider and our correctness criteria. Since we are interested in applications of BA, we select the failure assumptions that were used for BA: the n data processing nodes are either perfect or insane, and there are at most m insane nodes. (We would prefer to have only sane and insane nodes, but recall that BA, as well as reliable data processing, is not feasible in this environment.) We continue to make assumptions N1, N2, T1, T2, and T3. The assumptions for input/output nodes are discussed later.

Since data processing is considerably more complex than BA, we cannot give a precise definition of correctness. But, intuitively, we want the system to satisfy the following conditions. (In stating these conditions we have to strike a balance between what is desirable and what is reasonable to expect from a system.)

C1. *Users should obtain the same* results *from the system that they would obtain from an ideal system where no failures occur.* Specifically, if **T** is the set of transactions submitted to the (real) system, then there must be some serial schedule S of a *subset* of **T** that if executed on the ideal system would yield exactly the same results and would leave the database in the same state.

C2. *If a transaction T_i is submitted at a perfect input node, then T_i will be in the resulting schedule S.*

Note that the system does not necessarily have to process transactions submitted at faulty input nodes. However, if the system executes a transaction (even one from a faulty input node), it must be precisely what the user submitted. For example, if a user submits a transaction to withdraw 10 million dollars, either that exact operation is performed, or nothing is done. Withdrawing 9.9 million is not considered correct.

From the point of view of the users it would be preferable if the system guaranteed the correct execution of *all* transactions, but we feel this would place an unreasonable responsibility on the input nodes. At the other extreme, the design of the database system would be simpler if the users could tolerate some modifications in their transactions. This may be the case in some applications, but not in general. For instance, in a flight control application, the altitude and speed are read off sensors, and used to compute the settings of the throttle and wings. In such a case readings off the sensors (i.e., the input transactions) could be useful even if they were "slightly off." Even if we knew the application (which we do not), it would be very difficult to define precisely which deviations would be acceptable and which would not.

Hence we take conditions C1 and C2 as a reasonable compromise for a *general-purpose* data processing system. Users will have to cope with the fact that their transactions may not be executed. (For instance, if their terminal appeared dead, they would switch to another one. If a missile had to be fired, they would write their "fire missile" transactions so that any 2 of 3 would cause the missile to be fired.) On the other hand, the system guarantees[5] that what it does execute is correct.

Conditions C1 and C2 do not specify in any way the *time* at which a transaction will be executed (assuming it is executed). From the user's point of view, this is not acceptable. For example, consider a user who submits the transaction T_1 to withdraw 10 million dollars from an account. Clearly, the user would not be happy if T_1 were executed a year after it was submitted. Thus, within a reasonably short period of time, the user must know if T_1 was (or is being) executed or not. If it is not being executed, the user needs a guarantee that it will never be executed in the future. With this guarantee the user can then submit a second transaction T_2 to retry the same operation. (Without the guarantee, the system could execute both T_1 and T_2.) Thus we have the following condition.

C3. *The time to commit a transaction is bounded.* Suppose that transaction T_1 is submitted by a user at time α_0. Then the system will make a commitment on T_1; that is, it will decide whether or not to execute T_1 by time $\alpha_0 + \delta$, where δ is a constant provided by the system designers. The commit decision is irreversible.

Note that C3 does not give users a guaranteed *termination* time. The time at which a transaction completes depends on the load other transactions place on the system, and is hence very difficult to bound. Similarly, note that C3 leaves the system free to execute transactions in any relative order. So that in an ideal system T_1 may find sufficient funds in the account for the withdrawal, but in a real one it may not. This is acceptable as long as the resulting schedule is equivalent to *some* serial schedule.

Having defined our correctness criteria, we now discuss how such a system could be constructed, and what additional assumptions have to be made. We start by considering the placement of the database. Suppose we have a single copy located at node N_i. If N_i were insane, it could do whatever it wished with the database. The database could easily be ruined, and this would violate condition C1. Assuming that N_i is perfect is not reasonable either, so, clearly, we cannot have a single copy of the database.

The solution is to replicate the database at several nodes. Since there can be up to m insane nodes, we need at least $m + 1$ copies to ensure that at least one perfect node manages the data correctly. Unfortunately, we cannot tell which is the perfect node, so $m + 1$ copies are not enough. We actually need $2m + 1$ copies. This way we can identify the correct results as those coming out of a majority of the copies.[6] (The number of nodes n must be greater than or equal to $2m + 1$.)

In addition to having $2m + 1$ copies, it is also necessary that all perfect nodes execute exactly the same transactions, *in the same order*. Otherwise, the databases

[5] The "fine print" in this guarantee will be discussed in a moment.

[6] Although we do not discuss it here, we could also use a weighted majority scheme, where nodes that are less likely to fail would carry more weight. The votes could also be adjusted dynamically as failures occur. See [29, p. 149] for details.

Also note that if a clock synchronization protocol is used for assumption T1, there could be an additional constraint on the number of copies that are needed to tolerate m failures. Fortunately, there are protocols that can operate with up to m failures out of $2m + 1$ processing nodes [10, 20, 22].

at the perfect nodes may diverge, and it will not be possible to identify the correct ones by the majority mechanism. (Actually, the actions of the transactions can be executed in different order at each node, as long as the resulting execution schedules are all equivalent.)

The strategy of replicating processing and voting on outputs has been called the *state-machine approach* [19, 28], and is well known, especially in the construction of reliable hardware [27, 29]. What we are doing here is simply pointing out that the BA failure assumptions force us into this type of processing. We now search for applications of BA in this environment.

Incidentally, the operations we are dealing with here (i.e., the database transactions executed at each node) are higher level than what is typically used in redundant systems. We could just as well use lower level operations (e.g., disk reads and writes), but we feel that this would increase the cost of guaranteeing that the same sequence of operations is executed at each node. In other words, transactions are usually the most compact way to represent the operations on the database: a transaction is typically a call to a predefined program (e.g., "reserve a seat on flight 784") or a sequence of high-level relational commands. Of course, in cases where this is not true, the lower level operation may be more appropriate (for instance, if transactions must be described by large portions of code and if they access a minimal part of the database).

(We have also decided to do the database computations for a transaction at the same node where the data is located. This appears to be the best strategy for database transactions, but in cases where transactions must perform a lot of computing, relative to the amount of data they access, it may be more effective to move the computations to another node [28].)

Notice that this full replication environment is very different from conventional distributed database processing [13, 15, 26, 31, 33] because there is little communication between the nodes. The nodes must agree among themselves as to what the input transactions will be (discussed in subsection 4.2). The outputs of each transaction must also be compared in order to select the majority, correct one (discussed in subsection 4.1). But, unlike conventional processing, here nodes do not have to communicate to execute the transactions. A node does not have to request locks from the other nodes, and there can be no global deadlocks. There is no need to decide what node will execute a transaction, they all do. The updates made by a transaction do not have to be broadcast to other nodes: all (perfect) nodes will make the same updates automatically. (Another important difference between the two environments is, of course, cost (discussed in Section 4.6).)

4.1 Outputs

Consider a transaction T submitted to the system by a user. After T is executed at $2m + 1$ nodes, at least $m + 1$ will have the correct result, x. We now need some mechanism to convey this result to the user. The mechanism will obtain the $2m + 1$ results of T, select the majority value and display it, say, on the user's terminal. Following assumption R3, we call this mechanism an output node.

This output node is now a critical component, and cannot be insane. Otherwise it could invalidate the results of the processing nodes by conveying garbage to the users. Since we have assumed that nodes can either be insane or perfect, we are forced to dictate that the output node be perfect. However, this seems overly restrictive: so far we have only required that $m + 1$ out of $2m + 1$ processing nodes be perfect, yet now we are requiring *all* output nodes to be perfect.

There are basically two ways out of this dilemma. The first is to move the output node to the user's head. That is, the user could directly examine the $2m + 1$ results of his transaction and perform the majority operation himself. This seems unsatisfactory for two reasons: first, the burden of failure should be placed on the system rather than on the user; second, users themselves are not always perfect.

The other alternative is to relax the failure model for output nodes. Since an ouput node performs a very simple function, it is easy to see what failures it could tolerate. It could lose or modify the outputs of some processing nodes as long as the majority, correct results were not altered. It could also crash, giving the user his results later, or not giving them at all. Since the system makes no guarantees about the completion time of a transaction, the crash of an output node is acceptable; that is, output nodes can be sane, as discussed in Section 2. When a user submits a transaction and fails to get the output, he can submit a query (directing the output to a different device) to see if his transaction committed, and, if so, what the results were.

4.2 Inputs

An input node is also a critical component. It must take a single transaction or command and distribute it to the processing nodes. And, as with output nodes, this critical component cannot be eliminated. If we ask the user to submit his transaction several times, we are only moving the distribution function to the user.

Again, it is not reasonable to assume that all input nodes are perfect. However, in the rest of this subsection, let us temporarily make this assumption and briefly study how the input transactions would be distributed to the processing nodes. Understanding this process with perfect nodes will then make it easier to understand this process with faulty nodes. Then, in Sections 4.3 and 4.4, we will explore other failure models for input nodes.

To satisfy condition C1, the processing nodes must execute the same sequence of transactions. Since the input nodes are perfect, they can agree on this sequence without using BA. For example, each input node I_j ($1 \le j \le r$) can number the transactions it generates as $T_{j,1}$, $T_{j,2}$, $T_{j,3}$, ... The processing nodes will then process the transactions in the order $T_{1,1}$, $T_{2,1}$, ..., $T_{r,1}$, $T_{1,2}$, $T_{2,2}$, ..., $T_{r,2}$, $T_{1,3}$, ... All perfect processing nodes will receive identical transactions, and hence will not have to compare their inputs with each other in BA fashion.

A few observations about this solution are in order:

(i) If two conventional database systems are given identical sequences of transactions, they may still process them in different orders (due to the randomness of some internal states, for instance, the position of the disk heads with respect to the platters). In our environment this cannot be permitted, since all nodes must have the same "final" state. Therefore, all processing nodes must ensure that the resulting execution schedule is equivalent to the serial schedule

$T_{1,1}$, $T_{2,1}$, ... There are a number of straightforward techniques for doing this.

(ii) Conditions C2 and C3 are trivially satisfied. Actually, the bound δ of C3 is 0.

(iii) With the above scheme, transaction processing will proceed at the rate of the slowest input node. That is, transaction $T_{j,k}$ cannot cannot be processed until $T_{i,k-1}$ for all i have been processed. To avoid this problem, we can set up the following convention. Input nodes transmit one transaction, $T_{j,k}$, every Ω time units. If a processing node does not receive a transaction from I_j in Ω units, then $T_{j,k}$ is taken to be a null transaction. If I_j needs to transmit more than one transaction in Ω units, it numbers them $T_{j,k,1}$, $T_{j,k,2}$, ..., and they are all processed as a single transaction. This solution relies on the synchronized clocks that perfect input nodes have.

4.3 Lazy Input Nodes

In our first model for input nodes we assume that when a processing node receives a transaction from an input node, then the transaction is correct, in the sense that the transaction was indeed submitted by a user. However, an input node may fail to send the transaction to some or all processing nodes, may wait arbitrary amounts of time between transmissions, and may send transactions in any order. Let us refer to this type of input node as *lazy*. (A lazy node is not necessarily sane because it can send out messages in any order and at any time.)

To use this model, we must make an additional assumption. Note that an input node could hold a transaction arbitrarily long before broadcasting it, violating condition C3. Also, the input node could broadcast it promptly to an insane processing node but then die. The insane node could then wait arbitrarily long, broadcast this transaction, and have the perfect nodes execute it.

To avoid these problems, we make input nodes attach a timestamp to each transaction, giving its arrival time. This timestamp can then be used by the processing nodes to discard old transactions. We must assume that input nodes have synchronized clocks (assumption T1) and that timestamps on transactions reflect the true arrival time.[7]

With these assumptions it is possible to build a system that satisfies the correctness conditions, but BA must now be used to ensure that perfect nodes execute identical sequences of transactions. We now outline one possible solution.

Let us say that the processing nodes agree to perform BA every Ω time units.[8] Also, let t_r be an estimate of the time it takes an input node to process an incoming transaction. (If a node exceeds this time, its transaction will not be processed.)

(1) Each processing node collects transactions received from the input nodes.

(2) At time $i\Omega$ each node selects the transactions with timestamps between $(i-1)\Omega - (\tau + t_D + t_r)$ and $i\Omega - (\tau + t_D + t_r)$. (Recall that τ is the maximum clock drift and t_D is the guaranteed network delivery time.) After time $i\Omega$, transactions arriving with a timestamp less than $i\Omega - (\tau + t_D + t_r)$ are discarded.

[7] The input nodes could have a larger drift τ than the processing nodes. This would only increase the commit bound of condition C3. We do not consider this option here.

[8] Ω can have any value. It can be smaller or larger than $\tau + (m+1)(t_D + t_s)$, the duration of each BA.

(3) At time $i\Omega$ each processing node (acting as a general) broadcasts the selected transactions using BA. The start time of the BA algorithm is $i\Omega$. The messages of the algorithm can be distinguished from those of previous and future executions because only they deal with transactions with timestamps between $(i-1)\Omega - (\tau + t_D + t_r)$ and $i\Omega - (\tau + t_D + t_r)$.

(4) At time $i\Omega + \tau + (m+1)(t_D + t_s)$ the BAs complete. Each perfect processing node has the same set of transactions. These are ordered (say lexicographically, or by timestamp) and executed after the transactions of the previous cycle complete. (As discussed in Section 4.2, nodes must ensure that the actual execution schedule follows this ordering.)

With this strategy, δ, the bound required by condition C3, is the time for a transaction to reach the processing nodes $(\tau + t_D + t_r)$ plus the worst-case delay for the start of BA (Ω) plus the time to complete the agreement $(\tau + (m+1) \cdot (t_D + t_s))$.

4.4 Erratic Input Nodes

A lazy node must transmit valid transactions with valid timestamps. Suppose we relax this restriction. Let us say that an *erratic* input node acts as a lazy node, except that it can transmit an erroneous transaction T' (possibly with an erroneous timestamp) to some processing nodes, instead of T, the transaction actually submitted by the user. Let us assume that an erratic node will transmit a given erroneous transaction T' to at most q processing nodes.

To satisfy condition C1, the processing nodes can under no circumstances execute T' instead of or in addition to T. Thus the processing nodes are forced to use voting to identify a correct transaction like T. However, we need to increase the number of processing nodes in order to guarantee that an erroneous transaction never gets a majority of votes.

For example, suppose that m, the maximum number of insane processing nodes, is 2 and that the system has 5 nodes. If one of the perfect nodes receives T', the 2 insane ones could also say that they received T', and this erroneous transaction would be processed because it had a majority of votes. However, if we increased the number of processing nodes to 6 (without changing m), then the nodes would not have a majority. In general, if T' can be received by q nodes, we need to have at least $2(q+m)$ processing nodes to prevent an erroneous transaction from being processed.

Given that there are at least $2(q+m)$ processing nodes (or $2m+1$ if q is 0), then the algorithm for inputting transactions is almost identical to the one for lazy nodes. The only difference is in step (4), where now only transactions received at a majority of nodes are executed.

Our input node models form a hierarchy. The strongest model, that is, the one that makes the most assumptions, is the perfect model. As discussed in Section 4.2, if input nodes are perfect, no BA in needed. The lazy model makes fewer assumptions; and with it processing nodes must use BA to agree on the input transactions. The erratic model makes even fewer assumptions, but requires the addition of $2q-1$ processing nodes; and processing nodes must still use BA. The weakest model is the insane model; but with it condition C1 cannot be satisfied.

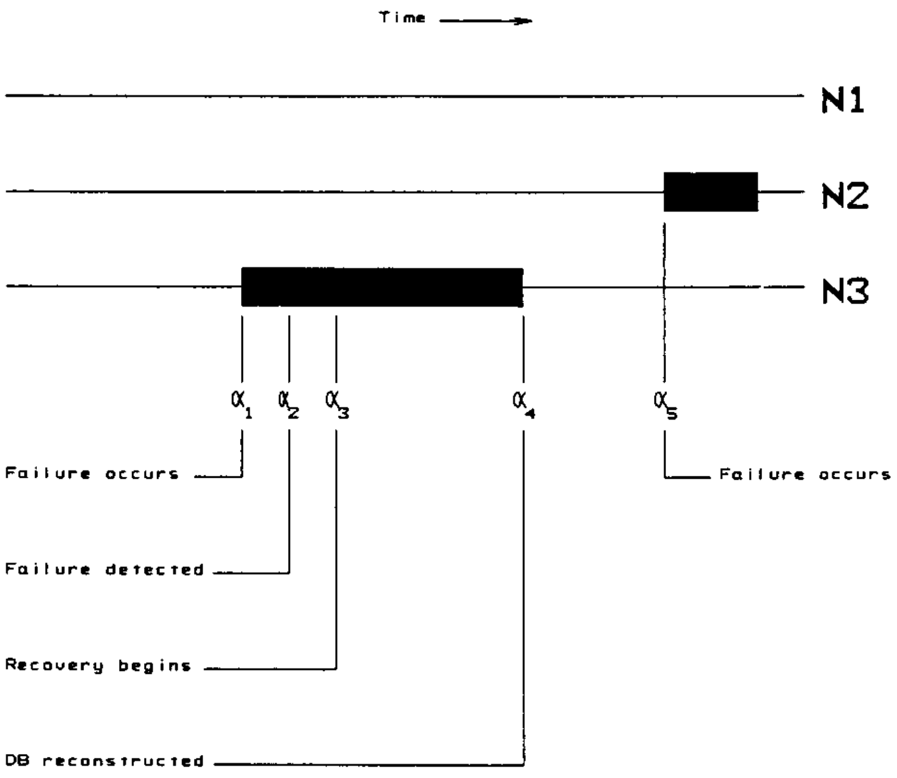

Fig. 3. Recovery from insanity.

4.5 Recovery From Insanity

Continuing our discussion of data processing with insane nodes, in this subsection we briefly address node recovery. Up to this point we have assumed that insanity is a permanent node property. However, it is desirable to repair insane nodes so that the system can then tolerate additional failures.

Figure 3 illustrates what we mean. There are 3 nodes in this system, so only a single failure can be tolerated. Suppose that N_3 fails at time α_1. From that point on its database may be ruined, and its results are not trustworthy. If we do not repair N_3, no failures of N_1 and N_2 can ever be tolerated. This is clearly not desirable.[9]

As discussed earlier (Section 3), the perfect nodes cannot be responsible for detecting an insane node like N_3. Instead, we must assume that N_3 detects its own failure. This is most easily accomplished by monitoring the outputs of all nodes. If its own output is different from that produced by a majority of the nodes, N_3 would identify itself as faulty. Alternately, the nodes could periodically compare their copies of the database. Of course, the comparisons could be made directly or through the use of database "checksums" or "signatures." In all cases, however, node N_3 must detect its own failure.

[9] [29, p. 216] and [32] study the mean time to failure of redundant systems without repairs.

By time α_2, the failure has been detected and N_3 must reconstruct its database copy. (Node N_3 may also have to repair other components, such as its clock or internal state tables, but we concentrate on the repair of the database itself.) Given that transactions are executed in timestamp order, the following steps can be used. First, N_3 selects a recovery time (α_3) and broadcasts it to the other nodes. (It does not matter whether the time is selected from the past or the future. However, for ease of discussion we assume the time is in the future.) At the appointed time, with regard to transaction timestamps, each perfect node takes a "snapshot" of their local database copy. (Only transactions with a timestamp prior to the recovery time are reflected in the snapshot.) These snapshots are used by N_3 to reconstruct its own database. During the snapshot exchange period, α_3 to α_4, the perfect nodes must continue processing transactions, even if at a reduced rate. (A single insane failure should never halt the system.) Therefore, N_3 must record any transactions that arrive after time α_3, postponing their processing until time α_4. After time α_4, N_3 must catch up to the other nodes in the system. At any time after α_4, even before N_3 is fully caught up, the system can tolerate a second failure, say at time α_5.

As with most algorithms, there are performance trade-offs associated with different implementations of detection and recovery. For example, hierarchical signatures could be used to identify portions of the database that have been corrupted, reducing the amount of data that must be exchanged during recovery. Additionally, the snapshots themselves may be implemented using many different methods, each with its advantages and disadvantages. A full examination of these problems is the focus of current research, and as such is beyond the scope of this paper.

4.6 Cost of Full Replication

The cost of full data and processing replication is high, so it must obviously be considered when a system is designed. Specifically, for each insane failure we wish to tolerate we must add to the system 2 processing nodes, each with a copy of the database.

This can be contrasted to the cost of tolerating a sane node failure. If availability is not important, no extra hardware is needed: when a node fails its data becomes inaccessible, but when it comes up the data will be correct. If availability is important, one spare node is needed per failure. Furthermore, the sane node algorithms usually have a lower message overhead.

Of course, the higher costs of full replication buy higher reliability, and both reliability and cost must be considered in making a decision. A full replication system can tolerate m insane failures, while even a single one of these failures could make a nonreplicated system yield incorrect results.

It is also important to note that full replication provides "comparable" protection against sane failures as a nonreplicated system (with equivalent hardware). That is, a properly designed full replication system with $2m + 1$ processing nodes can tolerate not m, but $2m$ sane node failures and still give users prompt access to correct data. (Recall that the BA algorithm that distributes transactions guarantees agreement among all perfect nodes, regardless of how many there are.) If the failed nodes stop after a failure, then it is still possible to identify the correct result (i.e., those at the operating sites). More than this number of sane failures makes the data unavailable but not incorrect. Thus the reliability that a full replication system can provide, with respect to sane failures, is comparable to (and not less than) that of a single node with $2m$ spares or backups.[10] Note that in general a system with n processing nodes can tolerate x sane failures *and* y insane failures and still make correct data available, as long as $n - x \geq 2y + 1$.

Full replication is expensive, but there are a number of ways to control its cost:

(1) A fast, local area network can be used to interconnect the nodes. This reduces the communication costs associated with BA.

(2) Only the critical parts of the database can be replicated. For example, in a bank database it may only be necessary to handle reliably the balances of accounts. Other data (e.g., customer addresses and credit history) could be handled with a single copy, thus reducing considerably the storage requirements. (One limitation: a transaction should not update critical data based on values of noncritical data.)

(3) If the replicated database is relatively small or if the network has large bandwidth, the crash recovery mechanisms (e.g., logging, shadow pages) for each copy can be eliminated. After a failure is detected the entire database can be copied from the other sites (see Section 4.5). This makes each processing node substantially more efficient for processing transactions during normal operation.

(4) Read-only transactions that are not critical can be processed as a single node. Again, this reduces the amount of work that has to be performed by the nodes.

(5) If all of the above ideas are combined, we can arrive at an "intelligent backup storage device" model. Here a large computer has a copy of the entire database and handles all transactions. A critical but relatively small part of the database is fully replicated at a number of smaller processors, the intelligent backup devices. These processors only have to execute the update transactions and the critical read operations, and do not keep any local recovery data. Under the right circumstances the processing load at each device will be relatively small, making it feasible to use inexpensive microprocessors. Thus the devices are similar to backup disks, except that instead of receiving commands to write blocks of data, they receive transactions and execute them themselves. Given the current cost of microprocessors, this approach seems to be an effective alternative to passive backup copies.

5. ANY OTHER USES OF BA?

So far we have studied data processing in an environment where (processing) nodes are either insane or perfect. Applying this failure model to critical components such as input/output nodes forced us to assume that they were perfect, hence there was no need for Byzantine agreement. This is because with data

[10] Note that lost messages where the loss is not detected by the sender (violation of assumption T2) must still be treated as an insane failure in the full replication system. Thus it could only tolerate m of these failures. A nonreplicated system could tolerate more of these failures by introducing indefinite delays: the sender keeps on transmitting a message until it receives an acknowledgment, and the receiver keeps waiting until it receives a message it is expecting.

processing we have a strict definition of correctness, where results that are "close" to the correct ones are not adequate. (However, in specific applications, where inputs are read off sensors or clocks, BA can be useful.)

We then relaxed the failure model for input/output nodes and identified one important application of BA: agreement of inputs from lazy and erratic nodes. Are there others?

We believe that the answer is no. In this environment all data and processing is fully replicated, and there is little interaction between the nodes. Once the nodes agree on the sequence of input transactions, there is nothing else they must do as a group.

It has been suggested that BA could be used to commit the transactions when they *complete* (as in conventional database systems), as opposed to committing them when they are submitted, as we have done so far. In this scenario nodes could process transactions in any order, and would constantly "vote" on which is the next transaction to complete. This way the transactions are committed in the same order, and the resulting schedules at all nodes would be equivalent.

The following example illustrates a serious flaw with this strategy. Suppose that m of the perfect nodes vote to commit transaction T, and the last perfect node, N_i, votes to abort. (For example, due to the order in which N_i executed the transactions, T is involved in a deadlock and must be aborted.) Without N_i we cannot commit T: we would not have the required $m + 1$ correct results. Thus processing nodes must abort a transaction whenever one or more nodes vote to abort. But if we do this, then the insane nodes will be able to paralyze the system by voting abort on all transactions! So, clearly, processing nodes should not wait until the end of a transaction to decide if it can be committed.

It has also been suggested that BA could decide upon the participants in the data processing algorithms. That is, using BA, nodes could agree on the set of processing nodes and on the set of input/output nodes. However, the data processing algorithms will work properly even if the nodes have different views of who is participating.

For example, consider a system with five processing nodes that is to grow to seven nodes. The system administrator can simply inform the nodes of this change. During the change, some nodes may think there are seven nodes, while others may think there are five. In this period those with a five-node view will ignore messages from the new nodes, and the new nodes in turn will think that those nodes are insane. Nevertheless, the original five nodes will function correctly, giving protection against two failures. When all nodes have a seven-node view, the system will tolerate three failures. Hence we see that BA is not required for changing the system participants.

5.1 BA with Sane Nodes

Finally, it has been suggested that BA could be used in a failure scenario where all nodes are sane. If processing nodes are sane, then we have what is considered conventional distributed data processing. Databases and transaction processing no longer need to be fully replicated. A single transaction may span several nodes. Internode concurrency control is needed, and a two-phase commit protocol [14] (or one of its variants) must be used to terminate transactions.

There are two ways in which BA protocols could be used in this environment. One is to assume that there are certain critical system operations (e.g., transaction commit, directory updating, or coordinator election) that require the higher reliability provided by BA protocols. In this case the system component that handles these operations is designed so that it can withstand insane node failures, while the rest of the system cannot. The data used by the critical component are replicated at all nodes, and the update "transactions" that are going to be processed against this data are handled by a BA protocol, as discussed in Section 4. (Thus, even though we are saying that a BA protocol is being used to elect a coordinator or to locate a file, the protocol itself is still being used to distribute inputs.)

There are a few potential problems with this approach. One is that the BA algorithms make no guarantee about the behavior of the failed nodes, and this may be unacceptable in a sane environment (i.e., where the database is not replicated). For example, if a BA algorithm is used to commit a transaction, a failed node is free to abort the transaction, even if the active nodes commit it. Of course, to avoid this problem, we may assume that the failed node is not really insane, but if this is done then a BA protocol may be unnecessary.

A second problem with this approach is that it may be an overkill. From the point of view of the system, it may be important to selectively protect certain components from arbitrary failures. Thus it may be valuable for the system administrator to know that the directory has an extremely low probability of being incorrect, even though the files it points to have a higher probability of being incorrect. However, from the point of view of the end-user of the database, this extra reliability may not be significant. That is, the end-user still has a single copy of his database, so a single insane failure will render it useless. The added robustness of, say, the directory is not very valuable: it is just as bad to get the valid location of an incorrect file as it is to get the invalid location of a correct file. In either case the user gets incorrect data. If the user really desires protection against insane failures, we should protect everything (as discussed in Section 4), and not just a few of the data processing steps: "A chain is as strong as the weakest of its links."

Note that we are not ruling out the use of BA protocols in a sane node environment. We are only stating that they must be used with caution, clearly identifying the users that require reliable data and protecting *all* of the data that these users need, directly or indirectly.

A second way to use BA protocols in a sane node environment is to modify them so that they take advantage of the sanity of the nodes. That is, if we know that the nodes are sane, then the BA algorithms can be simplified and the number of messages that have to be sent can be reduced. (For example, if a node receives a message from a second node saying that a transaction has committed, the node can immediately commit the transaction, without checking the veracity of the information.) We have absolutely no objection to this approach, and, as a matter of fact, the modified algorithms one obtains in this fashion are very similar, if not identical, to the "conventional" algorithms. Thus a modified BA commit protocol is very similar to a three-phase commit with termination protocol [30]. (A termination protocol decides the fate of a transaction in case the coordinator fails.) A modified BA algorithm used to elect a coordinator is very similar to "conventional" protocols [13]. The only question in such a case is whether a BA protocol without the original insane node assumptions and the code to cope with insanity should still be called a BA protocol.

6. CONCLUSIONS

In this paper we have studied BA algorithms and their application to distributed data processing. We presented a distributed processing system similar to the state machine approach, and discussed the assumptions and conditions that specified the desired correctness of the system. The main use of BA appears to be in the distribution of transactions in an environment where processing nodes are insane or perfect and where input nodes are lazy or erratic.

Our search of BA applications has lead us to conclude that uses outside of input distribution may be limited. However, these observations must, of course, be treated with caution since we have only addressed a single area: general-purpose, distributed database processing. We would expect, and indeed we know, that BA has other applications outside data processing. For example, BA may be useful (although not necessary) in clock synchronization algorithms and in the construction of sane or fail-stop nodes [27]. As discussed in Section 4, BA can also be used to process inputs from sensors or other unreliable sources. However, in all of these other areas, BA protocols seem to be used in the same way they were used in data processing (i.e., to distribute inputs to a set of replicated processors). Finally, we must also point out that we made a number of assumptions in this paper (N1, N2, T1–T3, R1–R3). Although we believe that these are only simplifying assumptions and do not change the essence of our conclusions, we have no formal way of proving this.

ACKNOWLEDGMENTS

Several useful comments and suggestions were made by Sheldon Finkelstein, Jim Gray, Leslie Lamport, C. Mohan, Fred Schneider, and Ray Strong.

REFERENCES

1. AGHILI, H., ET AL. A prototype for a highly available database system. Res. Rep. RJ-3755, IBM Research Laboratories, Jan. 1983.
2. BERLEKAMP, E. *Algebraic Coding Theory.* McGraw-Hill, New York, 1968.
3. BERNSTEIN, P. A. What can we expect from database theory? Invited talk, *ACM SIGMOD Conference* (May 1983).
4. DATE, C. J. *An Introduction to Database Systems,* vols. 1 and 2, Addison-Wesley, Reading, Mass., 1981.
5. DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Trans. Inf. Theor. IT 22* (Nov. 1976), 644–654.
6. DOLEV, D., AND STRONG, S. Polynomial algorithms for multiple processor agreement. In *Proceedings 14th ACM Symposium on Theory of Computing* (1982), ACM, New York, 401–497.
7. DOLEV, D., AND STRONG, H. R. Authenticated algorithms for Byzantine agreement. IBM Res. Rep. RJ3416, IBM Research Laboratory, Mar. 1982.
8. DOLEV, D., AND STRONG, H. R. Distributed commit with bounded waiting. In *Proceedings 2nd Symposium on Reliability in Distributed Software and Database Systems* (July 1982), 53–60.
9. DOLEV, D., REISCHUG, R., AND STRONG, R. Early stopping in Byzantine agreement. IBM Tech. Rep. RJ3915, June 1983.

10. DOLEV, D., HALPERN, J., SIMONS, B., AND STRONG, R. Fault-tolerant clock synchronization. *PODC Symposium* (Aug. 1984).
11. FISCHER, M., LYNCH, N., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. MIT/LCS/Tr-282. Sept. 1982.
12. FISCHER, M. J. The consensus problem in unreliable distributed systems (a brief survey). Tech. Rep. YALEU/DCS/RR-273, Dept. of Computer Science, Yale Univ., June 1983.
13. GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Comput. C-31*, 1 (Jan. 1982).
14. GRAY, J. Notes on database operating systems. In *Advanced Course on Operating System Principles*. Technical Univ., Munich, July 1977.
15. LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Netw. 2* (1978), 95–114.
16. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21* (July 1978), 558–564.
17. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.
18. LAMPORT, L. The weak Byzantine generals problem. *J. ACM 30*, 3 (July 1983), 668–676.
19. LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst. 6*, 2 (Apr. 1984), 254–280.
20. LAMPORT, L., AND MELLIAR-SMITH, P. M. Byzantine clock synchronization. In *PODC Symposium* (Aug. 1984).
21. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed data storage system. Xerox Res. Memo, Apr. 1979.
22. LUNDELIUS, J., AND LYNCH, N. A new-fault tolerant algorithm for clock synchronization. In *PODC Symposium* (Aug. 1984).
23. LYNCH, N., FISCHER, M., AND FOWLER, R. A simple and efficient Byzantine generals algorithm. In *Proceedings 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems* (1982), IEEE, New York.
24. MOHAN, C., STRONG, H. R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. Res. Rep. RJ-3882, IBM Research Laboratories, June 1983.
25. PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
26. ROTHNIE, J. B., JR., ET AL. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 1–17.
27. SCHLICHTING, R. D., AND SCHNEIDER, F. B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug. 1983), 222–238.
28. SCHNEIDER, F. Comparison of the fail-stop processor and state machine approaches to fault-tolerance. Tech. Rep. TR 82-533, Dept. of Computer Science, Cornell Univ., Nov. 1982.
29. SIEWIOREK, D. P., AND SWARZ, R. S. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass., 1982.
30. SKEEN, D. Crash recovery in a distributed database system. Ph.D. thesis, Memo. UCB/ERL M82/45, Univ. of California, Berkeley, May 1982.
31. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. SE-5* (May 1979), 188–194.
32. TAY, Y. C. Byzantine agreement in shortlived: Reliability of K-resilient distributed protocols. Tech. Rep. TR-18-83, Center for Research in Computing Technology, Harvard Univ., June 1983.
33. WILLIAMS, R., ET AL. R*: An overview of the architecture. In *Proceedings International Conference on Database Systems* (Jerusalem, June 1982).

Received June 1984; revised April 1985; accepted August 1985