Impossibility of Distributed Consensus with One Faulty Process*

Michael J. Fischer
Yale University
New Haven, Connecticut

Nancy A. Lynch
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

Michael S. Paterson
University of Warwick
Coventry, England

September 1982

Keywords: Asynchronous computation, commit, consensus, distributed computing, distributed database, and fault-tolerance.

©1982 Massachusetts Institute of Technology, Cambridge, MA. 02139

*This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

ABSTRACT

The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. We show that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

1. Introduction

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing. It is at the core of many of algorithms for distributed data processing, distributed file management, and fault-tolerant distributed applications.

A well-known form of the problem is the "transaction commit problem" which arises in distributed database systems [DS1, G, LS, La, Le, Li, R, RLS, S, SS]. The problem is for all the data manager processes which have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or to discard them. The latter action might be necessary, for example, if some data managers were for any reason unable to carry out the required transaction processing. Whatever decision is made, all data managers must make the same decision in order to preserve the consistency of the database.

Reaching the type of agreement needed for the "commit" problem is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a number of possible faults such as process crashes, network partitioning, and lost, distorted or duplicated messages. One can even consider more Byzantine types of failure [DS2, DLM, DFFLS, FL, LFF, LSP, PSL] in which faulty processes might go completely haywire, perhaps even sending messages according to some malevolent plan. One therefore wants an agreement protocol which is as reliable as possible in the presence of such faults. Of course, any protocol can be overwhelmed by faults that are too frequent or too severe, so the best that one can hope for is a protocol which is tolerant to a prescribed number of "expected" faults.

In this paper, we show the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death. We do not consider Byzantine failures, and we assume that the message system is reliable — it delivers all messages correctly and exactly once. Nevertheless, even with these assumptions, the stopping of a single process at

an inopportune time can cause any distributed commit protocol to fail to reach agreement. Thus, this important problem has no robust solution without further assumptions about the computing environment or still greater restrictions on the kind of failures to be tolerated!

Crucial to our proof is that processing is completely asynchronous, that is, we make no assumptions about the relative speeds of processes nor about the delay time in delivering a message. We also assume that processes do not have access to synchronized clocks, so algorithms based on timeouts, for example, cannot be used. (In particular, the solutions in [DS1] are not applicable.) Finally, we do not postulate the ability to detect the death of a process, so it is impossible for one processes to tell whether another has died (stopped entirely) or is just running very slowly.

Our impossibility result applies to even a very weak form of the *consensus problem*. Assume every process starts with an initial value in $\{0, 1\}$. A nonfaulty process decides on a value in $\{0, 1\}$ by entering an appropriate decision state. All nonfaulty processes which decide are required to choose the same value. For the purpose of the impossibility proof, we require only that *some* process eventually make a decision. (Of course, any algorithm of interest would require that all nonfaulty processes make a decision.) The trivial solution in which, say, 0 is always chosen is ruled out by stipulating that both 0 and 1 are possible decision values, although perhaps for different initial configurations.

Our system model is rather strong so as to make our impossibility proof as widely applicable as possible. Processes are modelled as automata (with possibly infinitely many states) which communicate by means of messages. In one atomic step, a process can attempt to receive a message, perform local computation based on whether or not a message was delivered to it and if so on which one, and send an arbitrary but finite set of messages to other processes. In particular, an "atomic broadcast" capability is assumed, so a process can send the same message in one step to all other processes with the knowledge that if any nonfaulty process receives the message, then all the nonfaulty processes will. Every message is eventually delivered as long as the destination process makes infinitely many attempts to receive, but messages can be delayed arbitrarily long and delivered out of order.

The asynchronous commit protocols in current use all seem to have a "window of vulnerability" — an interval of time during the execution of the algorithm in which the delay or inaccessibility of a single process can cause the entire algorithm to wait indefinitely. It follows from our impossibility result that every commit protocol has such a "window", confirming a widely-believed tenet in the folklore.

2. Consensus Protocols

A *consensus protocol* P is an asynchronous system of N processes ($N \geq 2$). Each process p has a one-bit *input register* x_p, an *output register* y_p with values in {b, 0, 1}, and an unbounded amount of internal storage. The values in the input and output registers together with the program counter and internal storage comprise the *internal state*. *Initial states* prescribe fixed starting values for all but the input register; in particular, the output register starts with value b. The states in which the output register has value 0 or 1 are distinguished as being *decision states*. p acts deterministically according to a *transition* function. The transition function cannot change the value of the output register once the process has reached a decision state; that is, the output register is "write-once". The entire system P is specified by the transition functions associated with each of the processes and the initial values of the input registers.

Processes communicate by sending each other messages. A *message* is a pair (p, m), where p is the name of the destination process and m is a "message value" from a fixed universe M. The *message system* maintains a multiset, called the *message buffer*, of messages that have been sent but not yet delivered . It supports two abstract operations:

send(p, m): places (p, m) in the message buffer;

receive(p): deletes some message (p, m) from the buffer and returns m, in which case we say (p, m) is *delivered*, or returns the special null marker ϕ and leaves the buffer unchanged.

Thus, the message system acts nondeterministically, subject only to the condition that if receive(p) is performed infinitely many times, then every message (p, m) in the message buffer is eventually delivered. In particular, the message system is allowed to return ϕ a finite number of times in response to receive(p) even though a message (p, m) is present in the buffer.

A *configuration* of the system consists of the internal state of each process together with the contents of the message buffer. An *initial configuration* is one in which each process starts at an initial state and the message buffer is empty.

A *step* takes one configuration to another and consists of a primitive step by a single process p. Let C be a configuration. The step occurs in two phases. First, receive(p) is performed on the message buffer in C to obtain a value $m \in M \cup \{\phi\}$. Then, depending on p's internal state in C and on m, p enters a new internal state and sends a finite set of messages to other processes. Since processes are deterministic, the step is completely determined by the pair $e = (p, m)$, which we call an *event*. (This "event" should be thought of as the receipt of m by p.) e(C) denotes the resulting configuration and we say that e can be *applied* to C. Note that the event (p, ϕ) can always be applied to C, so it is always possible for a process to take another step.

A *schedule* from C is a finite or infinite sequence σ of events which can be applied, in turn, starting from C. The associated sequence of steps is called a *run*. If σ is finite, we let σ(C) denote the resulting configuration, which is said to be *reachable* from C. A configuration reachable from some initial configuration is said to be *accessible*. Hereafter, all configurations mentioned are assumed to be accessible.

The following lemma expresses a "commutativity" property of schedules.

Lemma 1. Suppose that from some configuration C the schedules σ_1, σ_2 lead to configurations C_1, C_2 respectively. If the sets of processes taking steps in σ_1 and σ_2 respectively are disjoint, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C_3. (See Figure 1.)

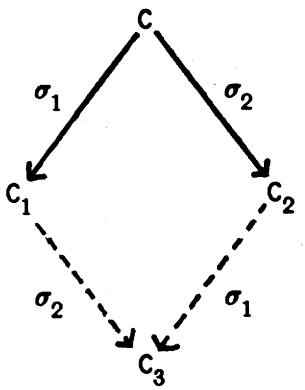

Figure 1.

Proof. The result follows at once from the system definition since σ_1 and σ_2 do not interact. □

A configuration C has *decision value* v if some process p is in a decision state with $y_p = v$. A consensus protocol is *partially correct* if it satisfies two conditions:

1. No accessible configuration has more than one decision value.
2. For each $v \in \{0, 1\}$, some accessible configuration has decision value v.

A process p is *nonfaulty* in a run provided it takes infinitely many steps, and is *faulty* otherwise. A run is *admissible* provided at most one process is faulty, and provided all messages sent to nonfaulty processes are eventually received.

A run is a *deciding* run provided some process reaches a decision state in that run. A consensus protocol P is *totally correct in spite of one fault* if it is partially correct, and every admissible run is a deciding run. Our main theorem shows that every partially correct protocol

for the consensus problem has some admissible run which is not a deciding run.

3. Main Result

Theorem I. No consensus protocol is totally correct in spite of one fault.

Proof. Assume to the contrary that P is a consensus protocol which is totally correct in spite of one fault. We prove a sequence of lemmas which eventually lead to a contradiction.

The basic idea is to show circumstances under which the protocol remains forever indecisive. This involves two steps. First, we argue that there is some initial configuration in which the decision is not already predetermined. Secondly, we construct an admissible run which avoids ever taking a step that would commit the system to a particular decision.

Let C be a configuration and let V be the set of decision values of configurations reachable from C. C is *bivalent* if $|V| = 2$. C is *univalent* if $|V| = 1$, let us say 0-*valent* or 1-*valent* according to the corresponding decision value. By the total correctness of P, and the fact that these are always admissible runs, $V \neq \phi$.

Lemma 2. P has a bivalent initial configuration.

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

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

Lemma 3. Let C be a bivalent configuration of P, and let $e = (p, m)$ be an event which is applicable to C. Let $\mathcal{C}$ be the set of configurations reachable from C without applying e, and let $\mathcal{D} = e(\mathcal{C}) = \{e(E) | E \in \mathcal{C}$ and e is applicable to E$\}$. Then $\mathcal{D}$ contains a bivalent configuration.

Proof. Since e is applicable to C, then by definition of $\mathcal{C}$ and the fact that messages can be

delayed arbitrarily, e is applicable to every $E \in \mathcal{C}$.

Now assume that $\mathcal{D}$ contains no bivalent configurations, so every configuration $D \in \mathcal{D}$ is univalent. We proceed to derive a contradiction.

Let E_i be an i-valent configuration reachable from C, i = 0, 1. (E_i exists since C is bivalent.) If $E_i \in \mathcal{C}$, let $F_i = e(E_i) \in \mathcal{D}$. Otherwise, e was applied in reaching E_i, and so there exists $F_i \in \mathcal{D}$ from which E_i is reachable. In either case, F_i is i-valent since F_i is not bivalent (by assumption) and one of E_i and F_i is reachable from the other. Since $F_i \in \mathcal{D}$, i = 0, 1, $\mathcal{D}$ contains both 0-valent and 1-valent configurations.

Call two configurations *neighbors* if one results from the other in a single step. By an easy induction, there exist neighbors C_0, $C_1 \in \mathcal{C}$ such that $D_i = e(C_i)$ is i-valent, i = 0, 1. Without loss of generality, $C_1 = e'(C_0)$ where $e' = (p', m')$.

CASE 1: If $p' \neq p$, then $D_1 = e'(D_0)$ by Lemma 1. This is impossible since any successor of a 0-valent configuration is 0-valent. (See Figure 2.)

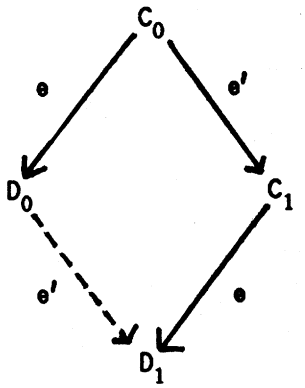

Figure 2.

CASE 2: If $p' = p$, then consider any finite deciding p-free run from C_0 with corresponding schedule σ, and let $A = \sigma(C_0)$. By Lemma 1, σ is applicable to D_i, and it leads to an i-valent configuration $E_i = \sigma(D_i)$, i = 0, 1. Also by Lemma 1, $e(A) = E_0$ and $e(e'(A)) = E_1$. (See Figure 3.) Hence, A is bivalent, which is impossible since A is univalent.

In each case, we reached a contradiction, so $\mathcal{D}$ contains a bivalent configuration. □

Any deciding run from a bivalent initial configuration goes to a univalent configuration, so there must be some single step which goes from a bivalent to a univalent configuration. Such a step determines the eventual decision value. We now show that it is always possible to run the system in a way that avoids such steps, leading to an admissible non-deciding run.

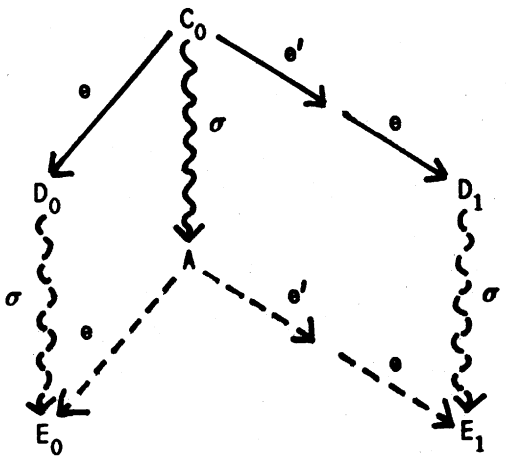

Figure 3.

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

Let C_0 be a bivalent initial configuration whose existence is assured by Lemma 2. Execution begins in C_0, and we ensure that every stage begins from a bivalent configuration. Suppose then that configuration C is bivalent and that process p heads the priority queue. Let m be the earliest message to p in C's message buffer, if any, and ϕ otherwise. Let e = (p, m). By Lemma 3, there is a bivalent configuration C' reachable from C by a schedule in which e is the last event applied. The corresponding sequence of steps defines the stage.

Since each stage ends in a bivalent configuration, every stage in the construction of the infinite schedule succeeds. The resulting run is admissible, and no decision is ever reached. It follows that P is not totally correct. □

4. Initially Dead Processes

In this section, we exhibit a protocol which solves the consensus problem for N processes as long as a majority of the processes are non-faulty and no process dies during the execution of the protocol. No process knows in advance, however, which of the processes are initially dead and which are not.

The protocol works in two stages. During the first stage, the processes construct a directed graph G with a node corresponding to each process. Every process broadcasts a message containing its process number and initial value and then listens for messages from L−1 other processes, where $L = \lceil (N + 1)/2 \rceil$. G has an edge from i to j iff j receives a message from i. Thus, G has indegree L−1.

In the second stage, the processes construct G^+, the transitive closure of G, in the sense that upon completion of this stage, each process k knows about all of the edges (j, k) incident on k in G^+. Each k also knows the initial values of all such j. After k discovers such an edge in G^+, we say that k *knows* about that edge and about the node j.

The computation of G^+ is carried out in the following way. First, each process broadcasts to all other processes its process number and initial value together with the names of the L−1 processes it heard from during the first stage. It then waits until it has received both the stage 1 and stage 2 messages from all its ancestors in G which it knows about. It initially knows only about the L−1 processes from which it heard directly during the first stage, but as it receives stage 2 messages, it may discover additional ancestors. Waiting continues until such time as all currently known about processes have been heard from.

At this point, each process knows all of its own ancestors and the edges of G incident on them, so it can compute all of the edges of G^+ incident on each of its ancestors. This enables it to determine which of its ancestors belong to an initial clique of G^+, that is, a clique with no incoming edges, for node k is in an initial clique iff k is itself an ancestor of every one of its ancestors. Since every node in G^+ has at least L−1 predecessors, there can be only one initial clique, it has cardinality at least L, and every process which completes the second stage knows exactly the set of processes comprising it.

Finally, each process makes a decision based on the initial values of the processes in the initial clique using any agreed-upon rule. Since all processes know the initial values of all members of the initial clique, they all reach the same decision.

The correctness of this protocol proves the following theorem.

Theorem II. There is a partially correct consensus protocol in which all nonfaulty processes always reach a decision, provided no processes die during its execution and a

strict majority of the processes are alive initially.

Acknowledgement

The authors would like to thank John Guttag for helpful discussions during the initial phase of this work, and Gene Stark for discussion of the results and a careful reading of the text.

References

[DFFLS] Dolev, D., Fischer, M., Fowler, R., Lynch, N. and Strong, R. Efficient Byzantine Agreement Without Authentication. In preparation.

[DLM] DeMillo, R., Lynch, N. and Merritt, M. Cryptographic Protocols. *Proc. 14th ACM Symp. on Theory of Computing* (1982), 383-400.

[DS1] Dolev, D. and Strong, R. Distributed Commit With Bounded Waiting. *Proc. 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems* (1982).

[DS2] Dolev, D. and Strong, R. Polynomial Algorithms for Byzantine Agreement. *Proc. 14th ACM Symp. on Theory of Computing* (1982), 401-407.

[FL] Fischer, M. and Lynch, N. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters 14*, 4 (1982), 183-186.

[G] Garcia-Molina, H. Elections in a Distributed Computing System. *IEEE Transactions on Computers Vol. C-31*, No. 1 (1982).

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

[La] Lampson, B. Replicated Commit. CSL Notebook Entry, Xerox Palo Alto Research Center (1981).

[Le] LeLann, G. Private communication, quoted in [La].

[Li] Lindsay, B. et al. Notes on Distributed Databases. *IBM Research Report RJ2571* (1979).

[LS] Lampson, B. and Sturgis, H. Crash Recovery in a Distributed Data Storage System. Xerox Palo Alto Research Center Manuscript (1979).

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

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

[R] Reed, D. Naming and Synchronization in a Decentralized Conputer System. Ph.D. Thesis, Massachusetts Institute of Technology. *Technical Report MIT/LCS/TR-205* (1978).

[RSL] Rosenkrantz, D., Stearns, R. and Lewis, P. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems 3*, 2 (1978), 178-198.

[S] Skeen, D. A Decentralized Termination Protocol. *(Proc. 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems* (1982) 27-32.

[SS] Skeen, D. and Stonebraker, M. A Formal Model of Crash Recovery in a Distributed Database Systems. *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks* (1981), 129-142.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 8/3/95

Report # LCS-TR-282

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 11(16-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAT: (1) UNFILMED TITLE PAGE	
	(2-10) PAGE 1-10
	(12-16) SCANCONTROL, PRINTER'S NOTES, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 8/3/95 Date Scanned: 8/18/95 Date Returned: 8/24/95

Scanning Agent Signature: Michael W. Cobb

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 8/18/1995

M.I.T. Libraries
Document Services