The Many Faces of Consensus in Distributed Systems

John Turek, IBM T.J. Watson Research Center
Dennis Shasha, New York University

Many distributed computing protocols require the ability to achieve consensus among processors. A parable of human communication starts this survey of the (im)possibilities.

Bob and Alice have discovered that they have a lot in common. For example, they both prefer e-mail to the telephone. On a cold winter day, Alice sends Bob electronic mail at 10 a.m., saying, "Let's meet at noon in front of La Tryste."

The e-mail connection between our two protagonists is known to lose messages, but today they are lucky and Alice's message arrives at Bob's workstation at 10:20 a.m. Bob looks at his calendar and sees that he is free for lunch. So he sends an acknowledgment.

Alice receives the acknowledgment at 10:45 a.m. and prepares to go out, when a thought occurs to her: "If Bob doesn't know that I received his acknowledgment, he might think I won't wait for him. I'd better acknowledge his acknowledgment."

And so it goes. We can show that, ultimately, neither Bob nor Alice will make it to La Tryste unless at least one of them is willing to risk waiting in the cold without meeting the other.

"The Parable of La Tryste" and the consensus problem

This parable holds several lessons for designers of distributed systems.

- *Easier problem lesson.* If the problem was simply that Alice wanted to be sure Bob received her original message, then the first acknowledgment would have sufficed. The issue is that Bob is not sure Alice knows that the first message arrived. Thus, the problem of transmission is easier than this problem of mutually coordinated action.
- *Reliable network lesson.* A phone call appears to solve the problem:

0018-9162/92/0600-0008$03.00 © 1992 IEEE

Alice: Let's meet at noon.
Bob: Sure, see you then.

The basis of this solution is the assumption that one party will hear what the other says within a bounded delay or that the existence of a problem will be evident within a bounded delay. If this assumption breaks down, either Bob or Alice might get stuck waiting out in the cold.

• *Probability lesson*. Imagine that Alice and Bob each send a flurry of duplicate messages instead of a single message each time. They might act on the assumption that at least one message will arrive. If they were right with probability p for each flurry, then a two-message protocol would succeed with probability p^2. Here, success means that neither would wait in the cold and they would lunch together at La Tryste.

The price of failure is higher in many applications. For example, if air traffic controllers used computers subject to these kinds of faults, we would be much more reluctant to use the airlines. For this reason, the probabilistic approaches cited in the literature on consensus eschew such risks. These approaches ensure that a decision will be made within a bounded amount of time with high probability. However, they insist that if a decision is eventually reached, it will be correct.

Consenting adults. In the consensus problem, a set of agents must all agree on a decision based on their initial states. Typically, only two decisions are allowed, 0 and 1. (Once a protocol for two decisions is available it can be extended to any number of decisions.) The numbers may represent actions. For example, 1 may represent "commit" and 0 may represent "abort" in a distributed database system. The agents must all output the same value and there must be some initial state for which 0 is the output and another for which 1 is the output.

Formally, a consensus protocol is correct if it meets the following conditions:

• *Consistency*. All agents agree on the same value and all decisions are final.
• *Validity*. The agreed-upon value must have been some agent's input.
• *Termination*. Each agent decides on a value within a finite number of steps.

In our parable, the *consistency* condition would be violated if it was possible for either Bob or Alice to wait outside in the cold alone. The *validity* condition would be violated if both Bob and Alice wanted to meet at La Tryste, but neither of them went. (This condition rules out the consistent but uninteresting solution where everyone always decides the same thing — for example, "Don't meet.") The *termination* condition would be violated if they were never to agree.

The right time and place. A prominent application of consensus is in commit protocols for distributed databases. In such protocols, all server sites must agree whether to commit or abort, and if any site wants to abort, then all sites must abort. The commit problem is strictly harder to solve than the consensus problem because of this priority in favor of aborts. Therefore, any result indicating the impossibility of consensus translates to an impossibility result for the commit problem.

A second important application area for consensus is ordered atomic broadcast protocols. Such protocols try to guarantee that if two messages, m and m', are sent, then either every working site will receive m first or every working site will receive m' first. As we will show, any system that can implement ordered atomic broadcast can also achieve consensus. Consequently, whenever consensus is impossible, so is ordered atomic broadcast.

In fact, consensus is part of any distributed system that embodies coordinated activity — from the synchronization of clocks, to the election of leaders, to the coordination of rocket firings.[1]

Moreover, consensus is closely related to fault tolerance. A system is *synchronous* if all processors proceed at predictable speeds. Otherwise, the system is *asynchronous*. A protocol is *wait-free* if no processor can indefinitely block the progress of any other processor. Herlihy[2] among others has shown that in an environment where n processors operate asynchronously, the ability to reach consensus among all the processors is a necessary condition for wait-free implementations of many shared-data structures and a sufficient condition for wait-free implementations of any shared-data structure. In other words, any asynchronous distributed system for which data sharing is important must be capable of consensus if it is to tolerate certain kinds of failures.

Because consensus is fundamental to so many distributed operations, its solution provides a fundamental building block to system designers.

"... begotten by despair upon impossibility."* Consensus can be easy or difficult to achieve depending on the kind of computer system (synchronous or asynchronous) and the failure assumptions. In a famous paper, Fischer, Lynch, and Paterson[3] showed the impossibility of deterministic consensus among two or more processors in an asynchronous distributed system. Since then, the consensus problem has been examined under many different synchrony and failure assumptions. For example, Fischer, Lynch, and Merritt[4] showed that consensus cannot be achieved in a synchronous environment if even one third of the processors are "maliciously" faulty — that is, if they act in a way that simulates an agent that tries to make the other processors make inconsistent decisions.

Given the role of consensus as a building block, these assumptions have a large impact on what can be achieved in practice. In this article, we survey known results regarding consensus, relating them to practice and explaining the small collection of elegant ideas embodied in their proofs. Our goal is to give practitioners some sense of the system hardware and software guarantees that are required to achieve a given level of reliability and performance. Our survey focuses on two categories of failures:

• *Fail-stop failures*. These occur when processors fail by stopping. While this is not a problem when processors are synchronous, the combination of asynchrony and fail-stop failures can make consensus impossible. We discuss these failures in the following section, "Hesitate and you're lost."

• *Byzantine failures*. These occur when processors fail by acting maliciously. This is a useful, though pessimistic, model of software failures. Depending on the number of failures in the system, consensus may be impossible under Byzantine failures even when the system is synchronous. We discuss these failures in the section titled "Plotting a Byzantine agreement."

---

*Andrew Marvell, "The Definition of Love," *The Poems of Andrew Marvell*, Hugh MacDonald, ed., Harvard Univ. Press, Cambridge, Mass., 1952, p. 34.

Hesitate and you're lost

A distributed system is made up of processors communicating through a shared communications medium, as illustrated in Figure 1. Sometimes we can assume that the communications media are reliable (for example, in backplane networks). Sometimes we can assume that processors are reliable (for example, in quadruple redundant hardware configurations). Suppose a given distributed system must solve problems at least as difficult as consensus. This would be true, for example, if the system was to include a distributed database. The designer should know how reliable the components must be in order to solve the consensus problem.

This section looks at how synchrony affects the spectrum of possibilities.

A world of (im)possibilities. Let's return to our original scenario where Bob and Alice are sending each other messages across a computer network. The difficulty in this scenario is that network delay has no bounds and messages can get lost. We observed that consensus is impossible under these constraints. Let's strengthen the network so that messages are never lost though they can still be delayed, and let's add the condition that either Bob or Alice could be fired at any time. Since the network never fails, Alice could send Bob a message and then wait for his response. But if Bob gets fired before he receives Alice's message, Alice may end up waiting indefinitely. Under these conditions, is consensus possible?

Fischer, Lynch, and Paterson[3] showed the surprising result that in a distributed system with an unbounded but finite message delay, there is no protocol that

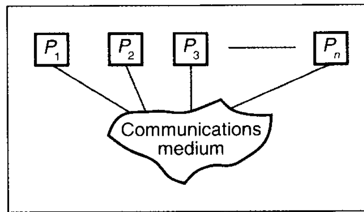


Figure 1. A distributed system.

can guarantee consensus within a finite amount of time if even a single processor can fail by stopping. This result implies no possibility of consensus for Bob and Alice under the redefined circumstances. (The reasoning behind this is discussed later under the subhead "Proving the impossible.")

While Fischer, Lynch, and Paterson's result shows that a completely asynchronous system cannot guarantee consensus, it does not give much sense of what can be achieved in practice. More optimistic assumptions on the timing constraints within the network and among processors can yield consensus protocols, even in the presence of multiple failures. Dolev, Dwork, and Stockmeyer[5] addressed this issue by identifying a set of system parameters for classifying asynchronous systems. The following items formally define a subset of those parameters:

- *Processors can be either synchronous or asynchronous.* Processors are synchronous if and only if there exists a constant $s \geq 1$ such that for every $s + 1$ steps taken by any processor, every other processor will have taken at least one step.
- *Communication delay can be either bounded or unbounded.* Delay is bounded if and only if every message sent by a processor arrives at its destination within t real-time steps, for some predetermined t.
- *Messages can be either ordered or unordered.* Messages are ordered if and only if processor P_r receives message m_1 before message m_2 when P_1 sends m_1 to P_r at real time t_1, P_2 sends m_2 to P_r at real time t_2, and $t_1 < t_2$. (For ordered atomic broadcasts, as described earlier in "The right time and place" and used below in case 2, the slightly weaker condition that either all sites either receive m' before m or all sites receive m before m' will suffice.)
- *Transmission mechanism can be either point-to-point or broadcast.* The transmission mechanism is point-to-point if a processor can send a message in an atomic step to at most one other processor. It is broadcast if a processor can send a message to all the processors in an atomic step.

Table 1 summarizes the possibilities for consensus presented by Dolev, Dwork, and Stockmeyer. In the system Fischer, Lynch, and Paterson studied, messages were unordered, communication was unbounded, and processors were asynchronous. As the table shows, consensus is impossible under these conditions. It is possible, however, in three minimal cases:

- *Case 1.* Processors are synchronous and communication is bounded.
- *Case 2.* Messages are ordered and the transmission mechanism is broadcast.
- *Case 3.* Processors are synchronous and messages are ordered.

We have included the third case for completeness. However, the best known algorithm for achieving consensus in this case requires an exponential number of messages and is therefore of little practical interest.

Case 1 describes the situation in which every processor can use time-outs to tell if another has failed. This assumption is the basis of the standard commit protocols that work under the fail-stop assumption, such as three-phase commit.[6]

Case 2 describes a situation in which processors can be asynchronous and some of them can also fail. However, they have an ordered atomic broadcast primitive (perhaps because they share a

Table 1. Conditions under which consensus is possible.

Processors	Message Order: Unordered		Message Order: Ordered		Communication
Transmission	Point-to-point	Broadcast	Broadcast	Point-to-point	
Asynchronous	No	No	Yes	No	Unbounded
Asynchronous	No	No	Yes	No	Bounded
Synchronous	Yes	Yes	Yes	Yes	Bounded
Synchronous	No	No	Yes	Yes	Unbounded

reliable bus). To achieve consensus, each processor broadcasts its initial value to all other processors. The processors then read messages from the network and note the first value received. Since messages are ordered, all the processors will agree as to which was the first value placed on the network.

A variation of case 2 is *k-casting*. This variation assumes that the transmission mechanism allows broadcast to at most k other processors. Dolev, Dwork, and Stockmeyer show that if k-casting is possible and messages are ordered, the system can achieve deterministic consensus in the presence of up to $k - 1$ failures.

Another variant assumes that processors are "nearly" synchronous. If a processor can read, process, and write to the network in one atomic step, the addition of bounded communication delay and broadcast transmission will be sufficient for achieving consensus. The idea is that if processors can execute a critical section of code within a predictable amount of time, then the problems associated with processor asynchrony can be overcome. This can often be achieved in practice by having processors disable interrupts during the critical code section.

Agreeing on shared memory. Does consensus become easier to implement in a system with a reliable shared memory? Intuition might suggest that the inherent broadcast capabilities and reliability of shared memory could suffice for consensus. While this is true for the Byzantine failures in synchronous systems that we will discuss later, it is not the case for asynchronous systems.

Herlihy[2] showed the impossibility of consensus in a distributed system with asynchronous processors and a shared memory that supports only reads and writes. Achieving consensus requires adding synchronization primitives to the shared memory. In fact, Herlihy showed the existence of a hierarchy of increasingly more powerful synchronization primitives that allow processors to achieve consensus in the presence of increasingly many faults.

To understand why shared memory is not enough, recall the minimum conditions presented by Dolev, Dwork, and Stockmeyer. Shared memory with read and write provides the equivalent of a broadcast mechanism, but does not offer the equivalent of ordered messages. Once two processors have written their messages to the shared memory, there is no way for a third processor to determine which one wrote its message first. Actually, because of the asynchronous nature of the processors, even two writing processors can't agree which wrote its message first.

```
fetch&add(m, v)
  begin /*Atomic action*/
    oldm ← m
    m ← m + v;
    return(oldm);
  end; /*Atomic action*/
```

Figure 2. Fetch&add (consensus number = 2).

```
compare&swap(m, new, old)
  begin /*Atomic action*/
    if (m = old) then
    begin
      m ← new;
      return (true)
    end
    else return (false);
  end; /*Atomic action*/
```

Figure 3. Compare&swap (consensus number = n).

Given an asynchronous shared-memory system prone to fail-stop failures, Herlihy defines the *consensus number* of a synchronization primitive. A primitive with a consensus number n can achieve consensus among an arbitrary number of processors even if up to $n - 1$ processors stop. By definition, a primitive with a consensus number $n - 1$, but not n, cannot simulate a primitive with a consensus number n (otherwise, it too would have consensus number n). Conversely, a primitive with consensus number n can simulate a primitive with consensus number $n - 1$.

For example, atomic read and write operations have a consensus number of 1, but not 2. Therefore, in a shared memory allowing only reads and writes, no deterministic algorithm can achieve consensus among two or more processors even if only one of the processors is allowed to fail.

Figure 2 presents fetch&add, a primitive that reads and increments a location from memory in one atomic step. Fetch&add has a consensus number of 2, but not 3. Therefore, adding it or a variant to the shared memory pushes the impossibility result out to three or more processors in the presence of two or more failures.

The notion of a *universal primitive* is important. Such a primitive has a consensus number of n for arbitrary n (that is, all but one of the processors can stop and consensus will still be reached). Figure 3 shows a universal primitive called compare&swap. It replaces the value in memory location m with *new* if and only if the old value in memory is equal to *old*. It is not difficult to see that compare&swap is universal. Assume that a specified memory location, m, has an initial value of $\perp$. Each processor, P_i, proceeds as follows:

(1) Write initial value to location $a[i]$.

(2) Compare&swap(v, $\perp$, i). (That is, attempt to replace the $\perp$ in location v with the processor ID.)

(3) Decide $a[v]$.

Only one processor, P, will succeed with the compare&swap. All processors will decide on the value that P places in v.

Herlihy's work shows that the compare&swap is a more powerful synchronization primitive for achieving consensus than are the test&set and the fetch&add, thereby dispelling a popular myth on the relative power of the latter two primitives. Of course, this does not preclude their usefulness; techniques such as combining can make them more efficient than compare&swap. It just turns out that there are certain things they cannot do.

Recall that a *wait-free* protocol is one in which no processor can be held up indefinitely by the actions, or failures, of other processors. Since consensus in the presence of an arbitrary number of failures cannot be achieved without using a universal primitive, it follows that there are computations that cannot be performed in a wait-free manner in a distributed system without a universal primitive. Herlihy showed that in the presence of a universal primitive, any computation can be performed in a wait-free fashion. Thus, the ability to achieve consensus is necessary for any general-purpose distributed system that purports to tolerate failures.

Proving the impossible. Here we describe the proof of Fischer, Lynch, and

Paterson's impossibility result.[3] Namely, in a completely asynchronous message-passing system (that is, one in which messages have unbounded but finite transit times), no deterministic consensus protocol can tolerate even a single processor failure.

The state of a system, denoted a *configuration*, is defined by the messages that have not yet been delivered to their destinations and the individual states (that is, program counter and internal memory) of the individual processors. If at some point in the computation either 0 or 1 can still be reached, the system is said to be in a *bivalent* state. Otherwise, the system is said to be in a *univalent* state. We say the system is 0-valent if 0 has been decided and 1-valent if 1 has been decided.

An *event*, *e*, is defined as the receipt of a message, *m*, by a processor. For the sake of generality, *m* may be an empty message. Since we assume the protocol is deterministic, the processors can be said to make decisions only when an event occurs. A sequence, or subsequence, of events is called a *schedule*. The proof shows that an adversary can keep the protocol going forever by slowing processors down or killing a single processor. Specifically, the following two lemmas prove the theorem:

- *Lemma 1.* There exists an initial configuration that is bivalent.
- *Lemma 2.* Given a bivalent configuration, there exists a nonempty schedule leading to another bivalent configuration.

Lemma 1 is best described by a variation on the bald man's paradox. A man with a full head of hair is not bald. A man with little or no hair is bald. A man can be either bald or not bald (for example, if he has 1,000 or more hairs, he is not bald; otherwise, he is bald). If we removed each hair one at a time from a man with a full head of hair, then we will reach a point where pulling one more hair will cause us to change our description of him. However, if he was wearing a hat and only, say, 999 strands of his hair were showing, it would be impossible to determine whether he was bald or not bald.

If all processors start with an initial value of 0, then the system must decide 0 to satisfy the validity condition on consensus. Likewise, if all processors start with an initial value of 1, then the

Processor		Inputs to processors					
P_1	$\Leftarrow$	0	1	1	...	1	1
P_2	$\Leftarrow$	0	0	1	...	1	1
P_3	$\Leftarrow$	0	0	0	...	1	1
.							
.							
.							
P_{m-1}	$\Leftarrow$	0	0	0	...	1	1
P_m	$\Leftarrow$	0	0	0	...	0	1
Decides		0	?	?	...	?	1

Figure 4. Initial inputs to processors and the resulting decisions.

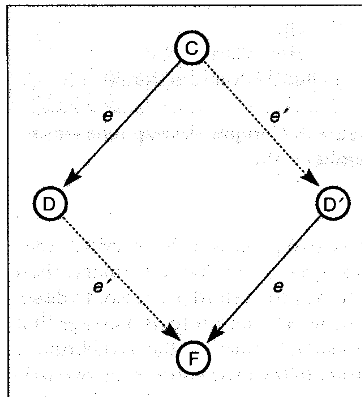


Figure 5. From one bivalent state to another.

system should decide 1. As shown in Figure 4, it is possible to go from a configuration in which all processors start with an input value of 0 to a configuration in which all processors start with an input value of 1 by flipping each processor's input value one at a time. Assume that there is no initial bivalent state. As with the bald man's paradox, there must be a single processor whereby flipping that input bit shifts the decision from a 0 to a 1. If an adversary caused the processor corresponding to that bit to fail before the protocol even began, then the two configurations would be impossible to distinguish from each other and would reach the same decision. This contradicts the assumption that one configuration could only have yielded a 0 and the other a 1.

To prove lemma 2, assume that the system is currently in a bivalent configuration, *C*. If a schedule exists that takes the system to another bivalent configuration, then we are done. Otherwise, since the system was in a bivalent configuration, there exist at least two events, *e* and *e*′, whereby *e* takes the system to a 0-valent configuration *D*, and *e*′ takes the system to a 1-valent configuration *D*′ (see Figure 5). No events lead to another bivalent state. Call e and *e*′ the *deciding events*. There are two cases:

- *Deciding events* e *and* e′ *occur on different processors.* Since events denote message receptions, applying *e* and *e*′ in either order yields the same configuration *F*. By assumption, if *e* is applied first, then *F* is 0-valent. If *e*′ is applied first, then *F* is 1-valent. This is clearly absurd. Hence, in a deterministic consensus protocol, any pair of deciding events yielding different valences must occur on the same processor.
- *Deciding events* e *and* e′ *both occur on some processor* P. If *e* occurs first and then *P* fails, the resulting configuration should be 0-valent. If *e*′ occurs first and then *P* fails, the resulting configuration should be 1-valent. But there is no perceivable difference between these configurations. Again, we get a contradiction.

Since an initial bivalent state exists and the adversary can keep the system in a bivalent state for an arbitrary period, there is no way of guaranteeing consensus in an asynchronous distributed system in which one processor can fail.

A different approach to understanding the issues and difficulties of the consensus problem uses a formalism called *knowledge logic*. A good reference to knowledge logic is the set of ACM conference proceedings from 1986 and 1988 entitled *Theoretical Aspects of Reasoning About Knowledge.*

Sharing messages. Herlihy proved that asynchronous processors, communicating via a shared memory, cannot achieve deterministic consensus in the presence of one faulty processor. He used a technique similar to the one used by Fischer, Lynch, and Paterson. Here, we relate the two results using a different kind of glue.

Even though consensus cannot be achieved in an asynchronous message-passing environment with faults or in an asynchronous shared-memory environment with faults, it would still seem that shared memory provides a more powerful primitive than message passing. In

one sense, this is true. Shared-memory systems can solve some problems even if a majority of the processors fail — problems that cannot be solved in a message-passing environment under the same conditions.

But what if fewer than half of the processors are allowed to fail? Attiya, Bar-Noy, and Dolev[7] have shown that under these circumstances the message-passing system of Fischer, Lynch, and Paterson can reliably emulate a shared-memory environment. This immediately lets us apply results from the Fisher, Lynch, and Paterson message-passing model to the read-write shared-memory model. Thus, the Fischer, Lynch, and Paterson impossibility result implies Herlihy's result and shows the impossibility of achieving consensus in the presence of even one fault in an asynchronous, read-write, shared-memory system.

One failure too many. The emulation result also provides an easier framework for implementing protocols in asynchronous message-passing systems, but before we show how to implement a shared memory, we will briefly discuss what we expect from a shared memory and why we cannot reliably emulate shared memory in a message-passing system in which a majority of the processors are allowed to fail.

To tolerate k failures, a system must maintain at least one copy of an object at $k+1$ different processors. Otherwise, the k failures could occur at the processors containing the copies of the object, and the object's value would be lost. However, maintaining at least one copy does not solve all our problems. Since the processors holding the copies may be slow to respond, two processors (or even the same processor) reading a copy of an object might not be reading the same copy. The fact that a writer may not have completed its write operation means that the later of two read operations may actually access an "earlier" version of the object. This leads to inconsistent executions.

The correctness criterion that we expect from a shared memory is the ability to implement *shared atomic registers*. An atomic register satisfies the following property: If processor P_1 finishes accessing the register before processor P_2 begins accessing the register and one of the accesses is a write, then P_2 reads or writes a "later" version than P_1. Specifically, assume that each value written into the register has a unique version number, then P_2 will see (write) a version number that is equal to or greater than that seen (written) by P_1.

To see why no algorithm could tolerate even half of the processors failing, consider a scenario in which the processors are partitioned into two groups of exactly equal size. Messages from one group to the other are "slow" while messages within each of the groups proceed at predictable rates. Given this scenario, processors in one group cannot distinguish between the situations in which all the processors in the other group are being slow or have failed. If the protocol assumes that the processors are slow, an adversary could cause the processors in the other group to fail. The protocol would not terminate and therefore would not be correct. If the protocol assumes that the processors in the other group have failed, then the two groups could come to different decisions, thus violating the consistency of the shared memory.

Two majorities always intersect. The critical problem in the previous subsection is that if the network can partition the set of processors, then two independent system components can proceed independently. Gifford[8] captured this observation in 1979 when he presented the idea of a quorum consensus. His algorithm shows how to reliably maintain several replicas of a data item in a synchronous distributed system prone only to fail-stop failures.

The idea is to make m copies of a data object X, $\{X_1, X_2, \ldots, X_m\}$. Writing proceeds by writing $w > k$ copies of X, where k is the number of failures that can be tolerated. This set of writes is called a *write quorum.* Reading proceeds by reading r copies of X. This set of reads is called a *read quorum.* The sum of the read and write quorums, $w + r$, must be greater than m to ensure an intersection between every pair of reads and writes.

Attiya, Bar-Noy, and Dolev[7] used this idea to show how to emulate a reliable shared memory in an asynchronous message-passing system in which fewer than half the processors can fail. To illustrate the algorithm, we first give an algorithm to emulate shared memory in a synchronous message-passing system. Associated with each copy is a version number. At any point in time, the copy (or copies) with the largest version number defines the current version. A read is executed as follows:

(1) Retrieve a read quorum of X.
(2) Select the copy with the largest version number.

A write is executed as follows:

(1) Retrieve the currently largest version number using the read procedure.
(2) Increment the version number.
(3) Send the new value along with the new version number to a write quorum.

The processors receiving the new value will replace the "old" value in their local memory if and only if the version number of the new value is larger than the version number of the old value.

Some care must be taken if multiple writers are allowed. To avoid confusion, all writers must write unique version numbers. We can guarantee this by concatenating the version number with the writing processor's ID.

While this algorithm for emulating the reading and writing of shared memory works well in a synchronous system, it will not work in an asynchronous system. The primary difficulty is the impossibility of guaranteeing that the copies will be read in the correct order. Figure 6 shows one such situation. There are three replicas — X_1, X_2, X_3 — of an object, X. A writer, W_1, could succeed in writing to X_1 before slowing down. A subsequent reader, R_1,

Time step	Processor 1 (W_1)	Processor 2 (R_1)	Processor 3 (R_2)
1	Write X_1		
2		Read X_1	
3		Read X_3	
4			Read X_2
5	Write X_2		Read X_3

Figure 6. Example showing how quorums can fail in asynchronous environments.

might read a quorum containing X_1 and X_3, thereby getting the new version of X written by W_1. Later, another reader, R_2, might read a quorum consisting of X_2 and X_3. This quorum does not contain the new version of X. R_2 therefore gets an earlier version than R_1, violating the conditions required for atomic registers.

Attiya, Bar-Noy, and Dolev get around this problem using a technique that turns out to be quite powerful in designing protocols for asynchronous distributed systems: *altruism*. Rather than being greedy and trying to complete its own operation as quickly as possible, each process acts altruistically. If it sees that some other process may not have completed its operation, it takes time out to help the process complete. In this case, the readers help the writer. When a reader reads a quorum and realizes that the writer did not finish its job, the reader plays the role of the writer and writes a quorum with the current value and version number. For this approach to tolerate k failures, the read and write quorum sizes will each be at least $k + 1$, and the system must have at least $2k + 1$ processors.

```
randcon(in)
  begin
    if in = 1
      globalcount ← globalcount + 1;
    else
      globalcount ← globalcount – 1;
    while –n < globalcount < n
    begin
      if globalcount < 0
        globalcount ← globalcount – 1;
      else if globalcount > 0
        globalcount ← globalcount + 1;
      else
      begin /*Atomic action*/
        if flip() = 1
          globalcount ← globalcount + 1;
        else
          globalcount ← globalcount – 1;
      end; /*Atomic action*/
    end;
    if globalcount > 0 decide(1);
    else decide(0);
  end;
```

Figure 7. Simplified algorithm for randomized consensus.

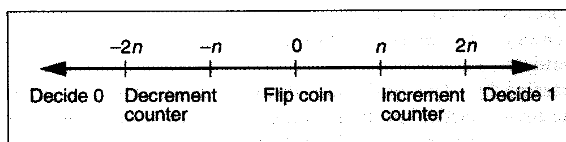


Figure 8. Regions for coin tosses in randomized consensus.

Foiling your adversary. We have already seen that deterministic consensus cannot be achieved in an asynchronous system in which even one processor is allowed to fail. Here we show that probability provides a powerful tool in this context. Each processor is allowed to flip a coin. The adversary cannot affect the result of this random coin toss, but in all other ways it remains unaffected. For example, it can still slow down processors at will. The algorithm we present guarantees both validity and consistency upon termination. Therefore, the adversary can only affect *when* the final decision is reached — not its correctness.

To simplify presentation, we show an algorithm from Aspnes[9] that works in shared memory. From the previous section, we know that any such algorithm can be converted into an algorithm that will function in a message-passing system. The algorithm takes its inspiration from a one-dimensional random walk, which brings us back to Bob and Alice.

Unable to agree on a meeting time with Alice, Bob consoles himself by going out drinking. He becomes intoxicated. His house lies at the end of the road on which the bar is located. Alice's house is about the same distance in the other direction. He is undecided whether to go home and sleep or go to Alice's house and chat. Assume that every time he takes a step he will stagger in the direction of his house or Alice's house with equal probability. If both houses lie n steps from the bar, how many steps will Bob take before reaching one of the two possible destinations?

The answer is on the order of n^2 (denoted $O(n^2)$). The walk provides us with the basis of the randomized consensus algorithm. Assume that processors can flip a coin and either add or subtract 1 from a global counter in one atomic step. Under this condition, Figure 7 shows the basic algorithm for randomized consensus on n processors.

Since the adversary has no control over the coin flips (or the order in which they are added to the global counter), the time required to hit one of the absorbing boundaries at either n or $-n$ corresponds to Bob's random walk. Once one of the boundaries has been reached, the remaining processors will eventually make the same decision.

To make the algorithm work even when it is not possible to flip a coin and increment the counter in one atomic step requires extending the region in which a coin can be flipped. Figure 8 shows these regions.

The proposed adversary is more powerful than what one would encounter in practice. In fact, the adversary will not maliciously adjust the speeds of processors. Rather, the speeds will be affected randomly. Aspnes and Herlihy[10] give an algorithm with the same running time as the algorithm in Figure 7, but theirs uses a *weakly biased coin* that will land on the same side at all the processors with high probability. Since the correctness of that algorithm is not particularly intuitive, we omit the details. In practice, the biased coin can be replaced by a shared table of "random" coin flips that the processors read to get the ith coin flip. With a failure model in which delays occur randomly, this modification to their consensus protocol yields an $O(n)$ algorithm.

Plotting a Byzantine agreement

Bob, Alice, and Joan are trying to get together for lunch. To simplify communication, they have decided to use a reliable medium — the telephone. Conference calling is not available, so at any

one time Bob can talk with either Alice or Joan, but not both. Mistrust and insincerity abound; however, at most one member of the trio is truly malicious (we do not know which one) and trying to make one of the other two wait in the snow.

Is there some protocol that the three can adopt such that (1) the two honest individuals will agree on whether or not to meet; (2) if all honest ones want to meet, then they will meet; and (3) if no honest ones want to meet, then they won't meet?

This problem is equivalent to the Byzantine generals problem studied by Lamport, Shostak, and Pease.[11] In their parable, several divisions of the Byzantine army are posted outside an enemy camp. Each division, headed by its own general, is trying to decide whether or not to attack the enemy camp. However, some of the generals are traitors and will try to keep the honest generals from reaching an agreement. A Byzantine failure is one in which a processor becomes traitorous and acts maliciously. The problem of reaching consensus in a distributed system prone to Byzantine failures is known as Byzantine agreement.

Byzantine failures were originally used to model hardware failures (or inherent flakiness) in avionics sensors, but they can also model software failures. If the software fails, we have no idea what it might do. Since it could do anything, the only fully general assumption to make is that it will do the worst thing possible. For it to do that, we assume that it is omniscient with respect to the state of the other (honest) processors.

This section discusses the conditions under which a synchronous distributed system can tolerate Byzantine failures.

Avoiding traitors. Given a synchronous message-passing system, is it possible to reach consensus in the presence of Byzantine failures? To answer this question, we need to be more specific regarding what the processors can do.

If Bob, Alice, and Joan were to make a conference call, then all three would hear the same message and it would be impossible for the traitor to lie to one person and not the other. Thus, Byzantine failures are not a problem under a communication medium that "broadcasts" messages to all the processors. Therefore, because of the inherent broadcast capabilities of shared memory, Byzantine failures do not constitute a serious problem in that environment. (The ability to verify the authenticity of messages partially simulates this broadcast ability and is discussed later under "Sign on the dotted line.")

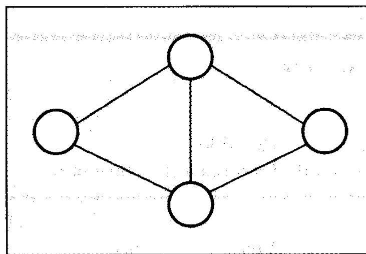


Figure 9. Graph with connectivity two.

When authentication is not available, Lamport, Shostak, and Pease show that Byzantine agreement is possible if and only if there are at least $3k + 1$ processors when k of the processors can fail. In other words, if one third or more of the processors are malicious, no deterministic algorithm can guarantee consensus among the honest processors. (We give their proof of this result under "The masquerade.")

When fewer than one third of the processors in a complete network are traitorous, deterministic agreement without authentication is possible. The solution given in Lamport, Shostak, and Pease requires a number of messages that is exponential in the number of individuals. Other researchers later showed that a polynomial number of messages will suffice for solving the problem under the same constraints.

Fischer, Lynch, and Merritt[4] extend the Lamport, Shostak, and Pease result to show that additional problems arise when the communication network is not complete. They define a graph's *connectivity* as the minimum number of nodes whose removal partitions the graph into two separate components. For example, Figure 9 shows a graph with connectivity two. The nodes represent processors, and the lines indicate communication between processors. A minimum of two nodes and their communications lines must be removed to partition the graph into separate components.

Fischer, Lynch, and Merritt showed that Byzantine agreement is possible if and only if the graph representing the communications network between the processors has connectivity greater than $2k + 1$, where k is the number of Byzantine failures that can occur. In other words, if removing half the individuals can partition the remaining individuals into two or more noncommunicating groups, Byzantine agreement will not be possible.

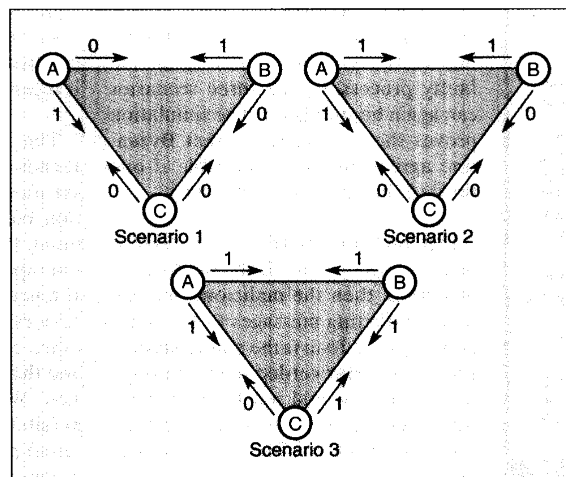


Figure 10. Scenarios leading to failure of Byzantine agreement.

The masquerade. When Joan decided to join Bob and Alice for lunch, without the ability to conduct a conference call, nobody could be sure who was honest. We first show that with three agents and at most one possibly faulty agent, the other two agents cannot agree on whether or not to meet. Intuitively, the difficulty is that Bob, assuming he is honest, cannot distinguish between the case where Alice is lying and the case where Joan is lying.

Fischer, Lynch, and Merritt give a simple proof of this idea. Suppose there was an algorithm that solved the problem at hand. Figure 10 illustrates three scenarios leading to the failure of any Byzantine agreement proto-

Table 2. Conditions required for consensus.

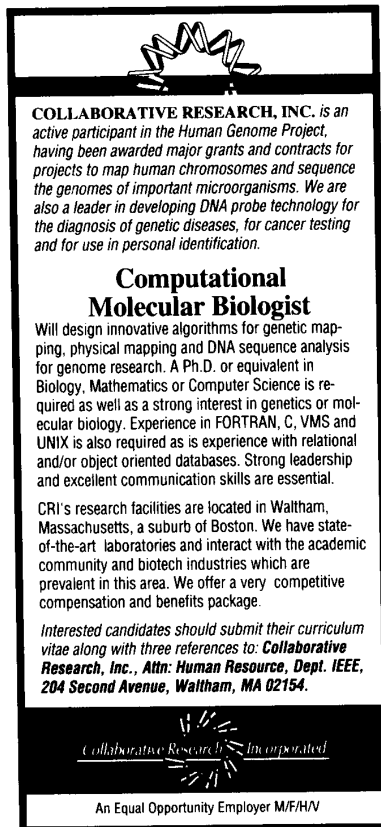
	Networks			
	Ordered Reliable Time-Bounded Broadcast	Reliable Time-Bounded	Reliable Unbounded	Unreliable
Processors never fail	Yes	Yes	Yes	No
Site failures Diagnostic time-out	Yes	Yes	No	No
Site failures No diagnostic time-out	Yes	No	No	No

col that does not use authentication. There are three agents, A, B, and C. In scenario 1, A is faulty. B and C start with the same input value, 0. B sees A starting with a value of 0, and C sees A starting with a value of 1. By the *validity* condition, the algorithm should ensure that B and C both decide 0.

In the second scenario, B is faulty, A starts with a 1, and C starts with a 0. If B sends the same messages to C as it did in the first scenario, C will see the same situation as in the first scenario. (We assume that in the first scenario A, the traitor, sent the same messages to C as in this scenario.) Therefore, the algorithm must once again decide 0.

The third scenario is one in which A starts with a 1, B starts with a 1, and C is faulty. If C sends the same messages to A as it did in the second scenario, then A sees the same situation as in the second scenario. (We assume that in the second scenario B, the traitor, sent the same messages to A as in this scenario.) Again, the algorithm must decide 0. However, the two nonfaulty processors both have an input value of 1, so the decision of 0 violates *validity*. This proves that consensus is impossible.

This result can be extended to an arbitrary number of processors by dividing the processors into three equally sized groups of processors. Allowing one of the groups to contain all the faulty processors, the three scenarios can again be simulated. The simulation proves the general result that Byzantine agreement is not possible if one third of the processors are faulty.



Sign on the dotted line. As we saw, if one of either Joan, Bob, or Alice is malicious, then the malicious one can send conflicting messages to the other two. Suppose Joan is the malicious one. Even if Alice forwarded Joan's message to Bob, Bob would not know if Alice was forging Joan's message or if Joan was being insincere. Therefore, he does not know whom to agree with.

However, if Alice forwards a photocopy of Joan's message, Bob can see that the writing is truly Joan's and will become immediately aware of the fact that Joan is the malicious individual. So he agrees with Alice. Joan is foiled.

This approach avoids problems because the traitorous agent can no longer send any message he or she wishes, since signatures cannot be forged. In computer systems, algorithms that guarantee that signatures are not corrupted are called *authentication* algorithms. Encryption provides the basis for authentication. Lamport, Shostak, and Pease give a simple authentication algorithm.

For the sake of simplicity, we assume the existence of a unique coordinator, C. When the coordinator is honest, all honest agents will output the coordinator's initial input. When the coordinator is dishonest, all honest agents will output a 0. The algorithm proceeds in $t + 1$ phases. Each message sent by a processor carries the signatures of all processors that have seen and transmitted the message. In phase i, there should be i signatures (in addition to the coordinator's) and no duplicates. That makes the message *legitimate*.

- *Phase 1.* The coordinator signs and sends an initial value to all agents. This constitutes their input. Note that the coordinator may send different initial values to different processors or may fail before sending messages to all processors.
- *Phase 2 through* $t + 1$. First, each agent signs and sends all legitimate messages received in the previous phase to all the processors. If the message is legitimate, then the agent records the value contained in the message.
- *At the end of phase* $t + 1$. An agent decides v if v is the only legitimate value it received. Otherwise, it decides 0.

The algorithm satisfies *termination:* It ends after $t + 1$ phases. The algorithm satisfies *validity:* If all processors function correctly and all have the same input, then they will agree on their initial input. The algorithm satisfies *consistency:* All correctly functioning processors will see the same values as all other correctly functioning processors and therefore will reach the same decision. With less than $t + 1$ phases, it is possible for an adversary to force different processors to reach different decisions.

Dolev and Strong[12] improved this exponential algorithm by noticing that old

messages do not have to be present. Their algorithm sends a number of messages that is quadratic in the number of processors.

In summary, for a group of processors to arrive at a common decision, they must solve the consensus problem. Distributed-system designers can save time by knowing the situations in which no algorithm is possible for consensus and those in which algorithms have already been discovered.

The fine line between impossibility and possibility trades processor reliability against network reliability. The more reliable the processors, the less reliable the network must be.

- In a synchronous distributed system with reliable message delivery and processors subject to Byzantine failures, consensus is possible as long as fewer than one third of the processors fail.
- In an asynchronous distributed system with reliable message delivery and processors subject to failure by stopping, consensus is not possible even if only one processor can fail. (Table 2 summarizes the conditions under which consensus is possible in different asynchronous systems.)
- In a synchronous distributed system in which messages can be dropped, consensus is not possible even if none of the processors fail.

Shared memory increases the reliability of the communications medium. It is essentially equivalent to adding a broadcast capability to a network. This avoids many of the problems created by Byzantine failures, but not the problems created by asynchrony. To solve these problems requires adding synchronization primitives such as compare&swap. The power of shared memory depends on the primitives it supports.

Finally, techniques such as randomization and authentication offer ways to overcome many impossibility results and often yield efficient algorithms.

Besides being useful, the consensus problem has resulted in many elegant impossibility proofs. These proofs teach a simple moral that we should all take to heart: Global knowledge is much stronger than local knowledge.

Or to put it in terms of our parable, Bob and Alice should ask to share an office. ■

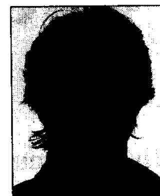
Acknowledgments

We thank Rajat Datta, Maurice Herlihy, and Farnam Jahanian for helpful discussions and the anonymous referees for their helpful comments. We also thank Bob and Alice for their help in the preparation (and presentation) of this article.

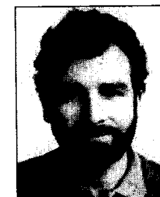
This research was partially supported by the National Science Foundation under Grants IRI-89-01699 and CCR-91-03953 and by the Office of Naval Research under Grants N00014-90-J-1110 and N00014-91-J-1472.

References

1. K. Birman, "How Robust Are Distributed Systems?" Tech. Report TR 89-1014, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1989.
2. M. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization," *Proc. Seventh Ann. ACM Symp. Principles of Distributed Computing*, ACM, New York, 1988, pp. 276-290.
3. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, Vol. 32, No. 2, Apr. 1985, pp. 374-382.
4. M. Fischer, N. Lynch, and M. Merritt, "Easy Impossibility Proofs for Distributed Consensus Problems," *Distributed Computing*, Vol. 1, Jan. 1986, pp. 26-39.
5. D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM*, Vol. 34, No. 1, Jan. 1987, pp. 77-97.
6. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.
7. H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message Passing Systems," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing*, ACM, New York, 1990, pp. 363-382.
8. D. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh ACM SIGOps Symp. Operating System Principles*, ACM, New York, 1979, pp. 150-159.
9. J. Aspnes, "Time- and Space-Efficient Randomized Consensus," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing*, ACM, New York, 1990, pp. 325-331.
10. J. Aspnes and M. Herlihy, "Fast Randomized Consensus Using Shared Memory," *J. Algorithms*, Vol. 11, No-3, Sept. 1990, pp. 441-461.
11. L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.
12. D. Dolev and H. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM J. Computing*, Vol. 12, No. 4, Nov. 1983, pp. 656-666.



John Turek is a research staff member at the IBM T.J. Watson Research Center. His research interests are parallel and distributed systems and algorithms. He obtained his BSc degree from MIT and his MS and PhD from the Courant Institute of Mathematical Sciences of New York University.



Dennis Shasha is an associate professor at New York University's Courant Institute, where he does research on transaction processing, real-time algorithms, and pattern matching. He also consults at Unix System Laboratories.

Shasha received his BS from Yale in 1977 and his PhD from Harvard in 1984. He has written a professional reference book, *Database Tuning: A Principled Approach* (Prentice Hall,1992), and two books about a mathematical detective: *The Puzzling Adventures of Dr. Ecco* (1988) and *Codes, Puzzles, and Conspiracy* (1992), both published by W.H. Freeman.

Readers can contact Turek at the IBM T.J. Watson Reasearch Center, PO Box 704, Yorktown Heights, NY 10598, e-mail turek@cs.nyu.edu, and Shasha at New York University, Courant Institute of Mathematical Sciences, 251 Mercer St., New York, NY 10012, e-mail shasha@cs.nyu.edu.