Fault-Tolerant Distributed Consensus

Lawrence Kesteloot

January 20, 1995

1 Introduction

A *fault-tolerant* system is one that can sustain a reasonable number of process or communication failures, both intermittent and permanent. The ability to solve the consensus problem is at the heart of any fault-tolerant system, because it is through consensus that non-faulty processes agree on which faults have occured, and how to circumvent them.

The *consensus problem* is defined as follows. All processes must agree on a binary value, based on the votes of each process. They must all agree on the same value, and that value must be the vote of at least one process (e.g., they cannot all decide on 1 when they all voted for 0). Note that it is not part of the definition that the decided value must be the majority vote; for example, in a distributed database application, each process may vote on whether to commit on a particular transaction, and if a single process votes *no*, then the decided value must be *no* and all processes must abort the transaction. There are variants of this definition, such as majority requirements or approximate agreement (agreement on numerical values where each process must decide on a value and all decided values must be within ϵ of each other).

Consensus in the presence of faults is difficult, especially when few assumptions can be made about the underlying system or the kinds of faults that can occur. Systems with different levels of synchrony or different kinds of failures require different algorithms. At one extreme, a system can be totally asynchronous, in that no assumptions can be made about the relative speeds of the processes or the communication medium. At the other, a system can be totally synchronous where we can assume upper bounds on processing and communication delays.

Usually, two kinds of failures are considered. *Fail-stop* failures cause a process to die at any time and stop participating in the algorithm. *Byzantine* failures are those where a process sends incorrect information, possibly according to a malevolent plan. Section 2 of this paper will consider fail-stop failures in asynchronous systems, and Section 3 will consider Byzantine failures in synchronous systems.

The other two combinations are not usually considered. Fail-stop failures in synchronous systems are fairly uninteresting: the failure of a process is immediately detected by the lack of messages from it, and all other processes can change their behaviors accordingly (see *early stopping* algorithms below). Byzantine failures in asynchronous systems are either equivalent to those in synchronous systems if the malevolent process sends messages, or equivalent to fail-stop failures if it does not.

This survey does not deal with tolerance of systemic failures (i.e., self-stabilization), but only with tolerance of process failures. Only message-passing systems are considered; shared memory algorithms fall in the category of wait-free constructions, which is outside the scope of this paper.

2 Fail-stop Failures in Asynchronous Systems

We start this section with an impossibility result, and spend the rest of it trying to modify either the system or our assumptions so as to make this kind of consensus possible.

2.1 Impossibility of Distributed Consensus with One Faulty Process

The title of this sub-section is that of the paper that proved this fundamental result in 1985 [9] by Fisher *et al.* Let us consider a system of completely asynchronous processes. That is, we can make no assumptions about the relative speeds of the processes or the speed of communication. We cannot check to see if a process has failed, and we have no reference to a clock to implement some kind of time-out mechanism. We do assume reliable communication (although messages may arrive in another order than they were sent) and that at most one process may fail (stop permanently) at any time. Despite these good conditions and benign failures, there is no algorithm which can guarantee consensus on a binary value in finite time.

This disturbing result is based on the fact that we cannot tell if a process has died or if it is just very slow in sending its message; we cannot distinguish between a process which is arbitrarily delayed and one which is indefinitely delayed. If this delayed process's input is necessary, say, to break an even vote, then the algorithm may be delayed indefinitely.

The most significant result of this paper is that totally asynchronous systems can never have any kind of fault-tolerance, since they cannot even handle the most benign of faults under the best of conditions. Fault-tolerance in asynchronous systems requires making some assumptions about the system or about the kinds of faults which can be handled. In real systems, this is usually done by assuming an upper bound in communication and processor speed, and considering a process faultly if it doesn't respond within a certain time. This allows the development of an algorithm in which, at every time-step, a process comes closer to making a decision (based on a received message or lack thereof).

The outline of the impossibility proof is as follows. If we can prove that there exists an initial state for which the final decision is undecided, and that starting at any undecided state can lead to another undecided state, then by induction we can show that the system can stay forever undecided. Even if we only consider fair runs (i.e., runs in which all processes have a chance to execute an action), we can show that there exists the possibility of a fair run staying undecided. This confirms the popularly held belief that any algorithm on asynchronous systems has a "window of vulnerability" during which a single process death could cause the algorithm to never terminate.

Since we do want fault-tolerance in asynchronous systems, then we must change our model in some way. The next three sub-sections discuss different ways that the model can be changed and the results of such changes.

2.2 Consensus with Probability 1

If we want consensus to be possible without changing the system conditions, then we must change the definition of "consensus". Instead of requiring guarenteed consensus in finite time, we may want to have consensus in finite time with probability 1. In other words, there is a chance that the algorithm will be indefinitely delayed, but the probability of this happening is 0.

Bracha and Toueg [3] described algorithms for both fail-stop and Byzantine failures which, if $n > 2t$ and $n > 3t$ respectively, led to consensus with probability 1. Their algorithm is probabilistic but does not incorporate randomization, as is done in other algorithms which have finite-time consensus with probability 1 [1].

For the fail-stop resilient algorithm, the key idea is that if there are n processes, t of which may be expected to be faulty, then a process can never expect more than $n - t$ acknowledgements from a broadcast. Each process then broadcasts its value and waits for $n - t$ answers. The problem now is to make sure that whatever decision made on those $n - t$ answers is the same as that made by another process, whose $n - t$ answers may have come from a different set of processes (since not all t processes may be faulty).

The algorithm proceeds in phases. Each process broadcasts its preferred value and the number of processes it has seen which also have this as their preferred value. (The latter number, the *cardinality*, is initially 1.) At each round, each process receives $n - t$ answers, each with a preferred value and cardinality. The processes then *change* their preferred value according to which value was preferred most by other processes. This continues roughly like this until a process, in a single phase, receives t messages of a single value each with at least cardinality $n/2$, a which point it knows that the algorithm will decide on that value in the next two phases and it can stop (but not before broadcasting enough messages for the next two phases).

Essentially, as the number of phases goes to infinity, the probability that consensus has not yet been reached goes to zero. Consensus with probability 1, although not as satisfying as guarenteed termination, is reasonable for most real systems.

2.3 Levels of Synchrony

Another way of achieving consensus in asynchronous systems is to relax the definition of "asynchronous" and allow some level of synchrony. The question then becomes, "How much and what kind of synchrony do we need before consensus is possible?"

We can identify three kinds of asynchrony in the system described by Fisher *et al.*: (1) process asynchrony, where each process may go to sleep for an arbitrary amount of time; (2) communication asynchrony, where there is no upper bound on the possible delay of a message; and (3) message order asynchrony, where messages may be delivered in a different order than they were sent. Dolev *et al.* in [6] investigate the possibility of a consensus algorithm with only some of these asynchronies in the system.

They prove that only making the processes synchronous is not enough, but making either the communication or message order synchronous is alone enough, so long as the idea of an "atomic step" of a processor is retained. (In Fisher *et al.*, each processor could perform an "atomic step", which consisted of receiving a message, performing some computation, and sending messages to other processes.) For example, if the atomicity of this operation is lost (i.e., if an arbitrary delay is allowed in the middle of the operation), then no consensus algorithm is possible with only communcation synchrony.

There are two components of an "atomic step" which are important: atomicity of a receive and send operation, and broadcast capability. Four minimal cases were described:

1. Process and communication synchrony.
2. Process and message order synchrony.
3. Message order synchrony and broadcast capability.
4. Communication synchrony, broacast cabability, and send/receive atomicity.

Any of these four scenarios is enough to allow the possibility of a consensus protocol, and any weakening of them makes such a protocol impossible.

2.4 Failure Detectors

One of the central assumptions about our asynchronous system is that we cannot detect the death of a process, and therefore we cannot distinguish a dead process from a merely slow one. Therefore, simply adding failure detectors would solve our problem without relaxing either our definition of consensus or our assumptions about the asynchrony in the underlying system.

We can imagine a module which is a failure detector for the system. This module could keep a list of processes which it thinks has crashed, and could regularly probe each process to update its list. Since this failure detector cannot be sure of a process death any more than any other process can, we should assume that it will not only make mistakes, but will make an infinite number of mistakes. The weakest properties that this module could have would be (1) all fail-stop processes are eventually detected, and (2) some correct process which is on the list of failed processes should eventually be taken off the list.

This can be implemented in practice by having the failure detector probe each process regularly; an unresponsive process p is placed on the list and a broadcast message is sent to all processes (including p) announcing its death. If p is has not actually crashed, then it will eventually refute its death announcement. Chandra and Toueg [5] show that this weak and unreliable model of failure detectors allows the consensus problem to be solved. The failure detector described above requires $f < n/2$, where f is the actual number of failures and n is the total number of processes. In [4], Chandra *et al.* prove that this failure detector is indeed the weakest that can solve the consensus problem. A stronger model of failure detectors allows $f < n$, although it may be difficult to implement in practice because it requires that at least one correct process is *never* suspected of being faulty.

3 Byzantine Failures in Synchronous Systems

In the last section, we saw that there is no totally correct consensus protocol for asynchronous processes. To have any kind of fault-tolerance, then, we must make assumptions about the system or the kinds of faults we can expect. We therefore assume a synchronous system of processes, in which we can detect the lack of a message through some means (for example, time-outs).

In what is probably the most famous use of anthropomorphism in distributed computing, Lamport *et al.* [12] introduced the paradigm of the Byzantine Generals Problem as a model for the consensus problem in light of faulty processes which send false messages.

The worst kinds of faults are those that follow a malevolent plan. They are the most disruptive because if there exists a certain scenario in which the consensus protocol will fail, we must assume that the malevolent processes will act out that scenario. If we can develop an algorithm to handle malevolent faults, then this algorithm will also be able to handle random, fail-stop, or any other kinds of faults.

We can discuss such an algorithm by using the Byzantine Generals Problem. A city is surrounded by Byzantine generals and their forces. The generals must agree on a plan of action; that is, they must agree whether to attack the city or to retreat. Some of these generals are traitorous and want to stop the loyal generals from coming to a consensus (or worse, make them each "agree" on a different plan). Any general can send a message to any other general directly, and can send oral or written messages through other generals. (Oral messages can be changed by intermediate generals, whereas written ones are signed and cannot be modified.) Messengers are reliable.

It is important to realize here that we are not trying to find out who is traitorous and who is not, but we are only trying to get the loyal generals to agree on a value.

The question now is, "How many traitorous generals are necessary before consensus among the loyal generals is impossible?" The question yields a different answer for oral and for written messages.

3.1 Oral Messages

With oral messages, if one-third or more of the generals are traitorous, then no consensus is possible. Let us start by formalizing what we want from a consensus algorithm. We want all loyal generals to come to a concensus on a boolean value. If each general could create a list of what he thinks all the generals (including himself) want to do, then he could simply run a function on that list, say taking a majority, and the output of this "majority" function would be his decided value.

Therefore we are now faced with the problem of having to make sure that every general gets the exact same list, so that every general can run the same "majority" function on his list and come up with the same value, thus coming to a concensus.

We would like our Byzantine Agreement algorithm to satisfy two conditions:

1. Every general should have the same idea of what every other general sent (i.e., every loyal general's list should be identical).

2. If general i is loyal, then every other general must have general i's true vote in his list. If we did not have this condition, then traitorous generals would be able to get the loyal generals to agree on a bad plan (one for which most or all loyal generals did not vote).

We can simplify the model a bit. It is easy to see that if we solve the problem for one sender general and $n - 1$ receiver generals, then we have solved it for n senders by simply running the one-sender algorithm n times. So as to not confuse the two models, we will speak of a commander sending orders to his $n - 1$ lieutenants. Any lieutenant may be traitorous, and the commander himself may be traitorous. We rewrite our conditions as follows:

$1'$. All loyal lieutenants must decide on the same order.

$2'$. If the commander is loyal, then all loyal lieutenants must all decide on his actual order.

To get an intuitive feel for the problem, consider one commander and two lieutenants, and one of the three is a traitor. If the commander is traitorous, he can send a "0" to one lieutenant and a "1" to the other. Since this would violate condition $1'$, our algorithm must do more. The obvious next step is to have the lieutenants compare orders to make sure the commander is not traitorous. Each lieutenant therefore sends his received value to the other lieutenant.

Let's look at it from lieutenant #1's point of view in two situations. In the first, lieutenant #2 is a traitor and lieutenant #1 gets a "0" directly from the commander and a "The commander said '1'," from lieutenant #2. In the second, the commander is a traitor and lieutenant #1 gets a "0" directly from the commander and a "The commander said '1'," from lieutenant #2. These two situations look the same to lieutenant #1. In the first situation, according to condition $2'$, he must choose the value "0". Since the two situations look alike to him, he will also choose "0" in the second situation. But in this second situation, lieutenant #2 will choose "1" (by symmetry), thus violating condition $1'$.

In fact, no amount of communication among the three people will allow the loyal lieutenant(s) to satisfy both conditions. Let us add one more lieutenant, but still only one of the four people is a traitor. In this new situation, less than one-third of the participants are traitors. Each lieutenant now gets three messages: one from the commander, and two from the other lieutenants. Since they are deciding on a binary value, then at least two of these messages will be the same, and the lieutenant can pick that value. To show that this value satisfies both conditions, we can do a case analysis.

If the commander is traitorous, he can send one value to one lieutenant and the other value to the other two. When the lieutenants exchange messages, they will each get the same list of three messages (since they are all loyal), and will agree on the same value. (Remember that condition $2'$ does not apply in this case.) If the commander is loyal, then the two loyal lieutenants will get the true order and can exchange it to confirm that theirs is the actual order. Even if the traitorous lieutenant lies about the received order, he cannot foil the other two since they will have a two-thirds majority on the true order.

If there are t traitors, then there must be at least $n = 3t + 1$ loyalists. This was shown in the above situation and is easily extended to any number of traitors. If t traitors are anticipated, then $t + 1$ rounds of information are exchanged. The first round has the format, "Here is my vote," and the second round has the format, "General i_1 said x," and the third round has the format, "General i_1 said that general i_2 said x," and so on. There is a total of $(n-1)(n-2)\dots(n-t-1)$ messages exchanged.

Yet, this limitation of having fewer than one-third of the generals be traitorous is bothersome; we would like to develop a system which is resilient to any number of traitors. We can achieve this with written messages.

3.2 Written Messages

With written messages, consensus is always possible (assuming, of course, that there are at least two loyal generals between whom to have a consensus). The model here is that when a general is relaying information, he can not send a message, but he cannot change a message. (Messages are signed and any change would detected by the receiving general.)

If we go back to our previous example of one commander and two lieutenants, a loyal lieutenant will always know who is a traitor. If he gets a "0" from the commander and a "The commander said '1'," or "The commander did not send anything," message from the other lieutenant, then clearly the commander is a traitor and some default value, say 0, is assumed by both lieutenants. If the loyal lieutenant does not receive anything from the other lieutenant, then that lieutenant is a traitor and the commander's order stands.

In practice, written messages are implemented with cryptographic techniques of authentication, such as RSA [13]. The number of messages stay the same as the case for oral messages, and as this number is exponential with n, we'd like to find an alternative algorithm which uses fewer messages.

3.3 Optimizing Byzantine Agreement

The Byzantine Generals Problem is a paradigm for distributed processes, some of which may be faulty. It might be fair to assume that most of the time, all processes will be correct (i.e., not faulty). An easy way to reduce the number of messages sent in a Byzantine Agreement algorithm would be to not run the algorithm at all if all of the processes are correct. We should then develop a Fault Detection algorithm which we run first, which would be weaker than the Byzantine Agreement algorithm. Specifically, either all processes are correct and they should come to a concensus, or at least one should detect the presence of a faulty process. Once a faulty process is detected, then the full-blown Byzantine Agreement algorithm can be run.

Hadzilacos and Halpern [10] showed that with oral messages, an average of $\lceil n(t+1)/4 \rceil$ messages are necessary for such an algorithm, where n is the number of processes and t is the number of expected faulty processes. With written messages, an average of $\lceil (n+t-1)/2 \rceil$ messages are necessary, which is considerably better.

Instead of considering the number of messages, we could consider the number of rounds. An algorithm which tolerates t faulty processes must run at least $t+1$ rounds in worse-case execution; however, if the actual number of faults f is less than t, then there may be a way for the algorithm to terminate earlier. This is referred to as *early stopping*, and was described by Dolev *et al.* in [8]. They show that if only fail-stop failures are allowed, then the minimum number of rounds is $\min(t+1, f+2)$, and they present an algorithm that achieves this bound. Since Byzantine failures are not considered, this result is somewhat weaker, but possibly useful in some systems where crashes are guarenteed to be the only faults.

Another approach is to use non-determinancy (i.e., probability). In [2], Bracha presents a non-deterministic algorithm with an expected number of rounds of $O(\log n)$, as long as $t = n/(3+\epsilon)$ for oral messages or $t = n/(2+\epsilon)$ for written messages. (The technique was later modified to yield a protocol with $O(\log n(\log\log n))$ expected rounds.)

The key is to use other Byzantine Agreement algorithms which have the following propreties: If t (number of faulty processes) is $O(n)$ then the algorithm is expenential in n, but if t is $O(\sqrt{n})$ then the expected number of rounds is constant. (There are several such algorithms, and our meta-algorithm will work with any "plug-in" algorithm with the above propreties.) If we can convert a system of n processes and $O(n)$ faulty processes into one with m processes and $O(\sqrt{m})$ faulty processes, then we can solve this new problem in constant time, leaving us only with the time to translate one system into the other.

Bracha shows that if we choose $m = O(n^2)$ and have each of these m processes be *compound processes* of $s = O(\log n)$ actual processes (each of which may be in several compound processes), then we can choose the compound processes in such a way that at most $\sqrt{m}$ of them are faulty. A standard $(t + 1)$-round Byzantine Agreement protocol can be run in each of the m compound processes (with an expected $O(\log n)$ number of rounds), and the result can be run on the m compound processes, with an expected constant number of rounds. This yields a total expected number of rounds of $O(\log n)$.

3.4 Eventual Byzantine Agreement

The original Byzantine Generals problem as stated by Lamport *et al.* [12] did not specify that all processes had to decide on a value simultaneously, although the algorithm given in [12] had that proprety. In [11], Halpern *et al.* discuss the possibility of an algorithm in which the processes can decide independently and possibly earlier. This is called Eventual Byzantine Agreement, to differentiate it from Simultaneous Byzantine Agreement.

Their results are based on a technique called *continual common knowledge*, which is a variant of common knowledge, i.e., the final result of Simultaneous Byzantine Agreement. The key is that at some point in the algorithm, some processor may have enough information to realize that the algorithm will eventually decide on a particular value. In other words, if common knowledge is that "Everyone knows that everyone knows that ... x," then *eventual common knowledge* is that "Everyone will know that everyone will know that ... x." It turns out that eventual common knowledge is too weak to solve the problem (because of the way Byzantine Agreement is done), so continual common knowledge (what everyone knows at any point in time) is used instead. They give a construction which can translate any Eventual Byzantine Agreement protocol P into an optimal protocol P'.

3.5 Approximate Agreement

The topic of approximate agreement is not strictly a problem of consensus, but is related and worth mentioning. We can modify the Byzantine Generals Problem so that each process inputs a real value rather than a binary value, and all processes must eventually decide on real values which are within ϵ of each other. Notice that if $\epsilon = 0$ then it is equivalent to the consensus problem with real values.

An algorithm described by Dolev *et al.* [7] works by successive approximation. That is, every round gets closer to the goal with a guarenteed convergence rate. The most interesting part of this algorithm is that it works both in synchronous and asynchronous systems (with $\epsilon > 0$). The conditions are similar to that of the consensus problem: all correct processes eventually halt with output values within ϵ of each other, and all decided values are within the range of input values of all correct processes. For synchronous systems, the convergence rate depends on the ratio between the number of faulty processes and the total number of processes. Convergence is guarenteed when $n > 3t$. For asynchronous systems, convergence is guarenteed when $n > 5t$.

The synchronous algorithm basically works as follows. Each process sends its value to every other process. If a faulty process does not send a value, then some default value, say 0, is chosen. Each process then runs the function $f_{t,t}$ on the n real values. Roughly speaking, the function f is chosen so as to eliminate the lowest and highest t values from the list and take the average of the rest; the faulty processes are thus unable to affect the convergence of the values. To show that it is necessary to have $n > 3t$, assume $n = 3t$. If all t faulty processes send a very high value to correct process p and a very low value to correct process q, then p and q will be taking their averages on two completely different sets of numbers, and they will not converge. However, if $n = 3t + 1$, then p and q will have one overlapping value at their median, and will (very slowly) converge towards each other.

The asynchronous algorithm is the same as above, except that each process only waits for $n - t$ messages, does not choose a default value, and runs the function $f_{2t,t}$ on the list. It is interesting to note that although exact consensus in a totally asynchronous system is impossible, ϵ can be chosen to be arbitrarily small to get as close to consensus as desired.

4 Conclusion

Although distributed consensus plays a key role in fault-tolerance, the kinds of algorithms we can use to solve this problem depend a great deal on the type of distributed system on which the protocol will run and the assumptions we make about the kinds faults to expect.

References

[1] Ben-Or, M. "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *ACM Symposium on Principles of Distributed Computing*, 1983, 27–30.

[2] Bracha, G. "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Protocol," *J. ACM 34*, 4 (October 1987), 910–920.

[3] Bracha, G. and Toueg, S. "Asynchronous Consensus and Broadcast Protocols," *J. ACM 32*, 4 (October 1985), 824–840.

[4] Chandra, T. D., Hadzilacos, V., and Toueg, S. "The Weakest Failure Detector for Solving Consensus," *ACM Symposium on Principles of Distributed Computing*, 1992, 147–158.

[5] Chandra, T. D. and Toueg, S. "Unreliable Failure Detectors for Asynchronous Systems," *ACM Symposium on Principles of Distributed Computing*, 1991, 325–340.

[6] Dolev, D., Dwork, C., and Stockmeyer, L. "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM 34*, 1 (January 1987), 77–97.

[7] Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. "Reaching Approximate Agreement in the Presence of Faults," *J. ACM 33*, 3 (July 1986), 499–516.

[8] Dolev, D., Ruediger, R., and Strong, H. R. "Early Stopping in Byzantine Agreement," *J. ACM 37*, 4 (October 1990), 720–741.

[9] Ficher, M. J., Lynch, N. A., and Paterson, M. S. "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM 32*, 2 (April 1985), 374–382.

[10] Hadzilacos, V. and Halpern, J. Y. "Message-Optimal Protocols for Byzantine Agreement," *ACM Symposium on Principles of Distributed Computing*, 1991, 309–323.

[11] Halpern, J. Y., Moses, Y., and Waarts, O. "A Characterization of Eventual Byzantine Agreement," *ACM Symposium on Principles of Distributed Computing*, 1990, 333–346.

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

[13] Rivest, R. L., Shamir, A., and Adleman, L. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM 21*, 2 (February 1978), 120–126.