

Illustrating the impossibility of crash-tolerant consensus in asynchronous systems

Felix C. Freiling

Department of Computer Science
University of Mannheim, Germany
freiling@informatik.uni-mannheim.de

Hagen Völzer

Institute for Theoretical Computer Science
University of Lübeck, Germany
voelzer@tcs.uni-luebeck.de

ABSTRACT

This exercise shows how a simple restricted algorithm can be used to present an introductory discussion on the crash-tolerant consensus problem in asynchronous distributed systems. This text is intended to support teaching the consensus problem in courses on distributed systems.

1. INTRODUCTION

Reaching consensus among a group of processes is a central problem in fault-tolerant distributed operating systems. State machine replication, database atomic commit, and reliable broadcast all include some form of consensus. The consensus problem has attracted a lot of research attention not only because of its practical importance but also because of its theoretical difficulty [1,8]: In fact, a fundamental result [6] states that it is impossible to solve consensus deterministically without any timing assumptions (in so-called *asynchronous* systems) if a single process may fail by crashing. Different concepts, such as partial synchrony [5], failure detectors [4], randomization [2], and input vector restrictions [7] have been proposed to circumvent the impossibility.

An intuition for that fundamental impossibility can be gained by studying its proof [6,9] and the various ways to circumvent it. However, the corresponding algorithms and proofs are not only rather involved, they are also discussed in different contexts using different terminologies and system models.

This exercise presents a lightweight alternate way to gain an intuition for the consensus impossibility. It is based on a minimal example, a very simple novel consensus algorithm that tolerates a single crash. That algorithm can be used to discuss all relevant concepts within a single framework. Our experiences with students in our distributed systems classes have encouraged us to bring this example to a wider audience to explain the different parameters of the system model which influence solvability of consensus and other types of agreement problems in asynchronous fault-tolerant distributed computing.

Throughout this paper processes communicate through reliable message passing, i.e., a message sent is eventually delivered and it is not altered or multiplied during transmission.

2. A SIMPLE CONSENSUS ALGORITHM

In the (binary) crash-tolerant consensus problem a set of distributed processes is required to reach a common decision. Initially, every process proposes a value (either 0 or 1). An

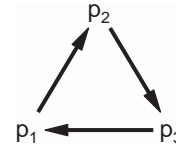


Figure 1: An oriented ring of three agents

algorithm which solves the consensus problem must then guarantee the following properties:

- Termination: Every process that does not fail (also called *correct*) eventually decides some value.
- Agreement: No two processes decide different values.
- Validity: The decided value must have been proposed by some process.

The problem is easy to solve under the assumption that no faults occur (some process broadcasts its value on which all others decide). The types of faults we consider here are unannounced crash faults where a process permanently stops executing steps. Note that, without the validity requirement, the problem has a trivial solution: Every process simply could decide 1 without any communication. The problem becomes interesting when validity is required and crashes may occur.

We now give a consensus algorithm for three processes that operates under the assumption that at most one process crashes. The problem the algorithm solves is slightly weaker than stated. We are only concerned with guaranteeing that some process eventually decides. An algorithm satisfying this weaker form of termination can be extended to an algorithm satisfying the stronger form above by adding an action that sends, just before a local decision, *decide* messages to all processes, which then trigger a decision at the receiving processes.

We assume that the processes are arranged in a ring in which they have a consistent sense of direction (see Fig. 1). The clockwise and counterclockwise neighbor of a process are called its *successor* and *predecessor* respectively. Our algorithm works as follows.

Each process has always an *estimate* of what the final decision value will be, which is initially its proposal value. Each process starts by sending its estimate to its successor and then waits for receiving a value.

Upon receipt of a value from its predecessor, a process compares the value with its current estimate. If they are

```

1 procedure consensus( $w \in \{0, 1\}$ ):  $\{0, 1\}$ 
2   var  $e : \{0, 1\}$  init  $w$ 
3   begin
4     send  $\langle e \rangle$  to successor
5     loop
6       wait until
7         upon receive  $\langle v \rangle$  from predecessor do
8           if  $e = v$  then return  $e$ 
9           else send  $\langle \text{CHG} \rangle$  to predecessor
10          upon receive  $\langle \text{CHG} \rangle$  from successor  $\wedge \mathcal{D}$  do
11             $e := f(e)$ 
12            send  $\langle e \rangle$  to successor
13          end wait
14        end loop
15    end

```

Figure 2: A simple consensus algorithm with a boolean parameter \mathcal{D} and a function f that either returns 0 or 1.

equal, the process decides that value, otherwise it returns a *change message* to its predecessor.

When a process receives a change message from its successor, it inverts its estimate, sends the new estimate back to its successor and waits again for a message.

This algorithm is shown as pseudo-code in Figure 2. Consensus is implemented as a procedure for each process p , which takes the proposal value of p as an argument and which returns the decision value of p . At the **wait-until** statement, a process blocks until one of the subsequent **upon** clauses can be executed. If both clauses can be executed at the same time, the process chooses a clause nondeterministically.

In line 10, there is a placeholder \mathcal{D} for an additional boolean condition. For now, we assume that \mathcal{D} is always *true*. The function f in line 11 inverts a value, i.e., $f(e) = 1 - e$. This function will later be redefined.

3. CORRECTNESS

3.1 Basic properties

First we consider some basic properties of the algorithm. If all three processes start with the same proposal value, it is easy to see that all processes that do not crash will quickly decide that value. This is true irrespective of a possible crash and therefore we have proven already the validity requirement.

If one of the processes is initially dead, the remaining pair of processes will come to a decision after at most three rounds of communication. We will later see that this is also true when that one crash happens at a later time: Once a crash has happened, a decision will be reached within three rounds of communication.

Finally, consider a scenario where all three processes start at the same time and proceed at the same speed and assume that all message delivery delays at all three channels are constant and equal. Because there is at least one pair of processes with the same proposal value, one of the processes decides when it receives its first message.

3.2 Agreement

We now show that our algorithm satisfies the agreement property, i.e., the decision value is unique regardless of proposal values, crashes and relative speeds of the processes.

The main intuition for the agreement property is that whenever a process p sends its estimate to its successor q , it promises q not to change its estimate until it receives q 's permission to do so, i.e., a change message. We say that p 's estimate is *locked* in a global state if that estimate does not change in any continuation from that state. The estimate of a process that decides is clearly locked.

Claim. If a process p decides v then v is the locked current estimate of p 's predecessor.

PROOF. A process can change its estimate only upon receipt of a change message. It is easy to verify that there is always at most one message in transit between each pair of neighbors. A process p decides v only upon receiving v from its predecessor q . From the time q has sent its estimate until its receipt by p , there can be no change message in transit from p to q . After p has decided, it will not send any change message to q either. Hence the claim. \square

Consider now two processes p and q that decide in the same execution. Suppose that p decides first. Process p is either q 's predecessor or q is p 's predecessor. In both cases, it follows from the claim above and the fact that a process always decides its own, locked current estimate, that both processes must decide for the same value.

Note that validity and agreement are satisfied regardless of how \mathcal{D} is instantiated, i.e., setting \mathcal{D} to *false* or *true* at any time cannot lead to a violation of validity or agreement. Similarly, the choice of the function f in line 11 does not affect validity and agreement.

3.3 Termination

Fischer, Lynch, and Paterson [6] have shown that no algorithm that tolerates a single crash solves consensus in an asynchronous system and therefore, our algorithm cannot guarantee termination without further assumptions. The algorithm has a non-terminating execution, which is illustrated in Fig. 3 (cf. also Fig. 1) and which we now explain.

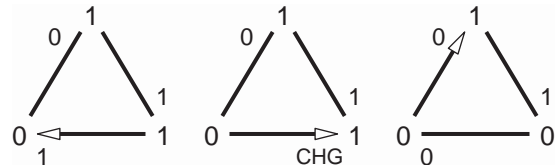


Figure 3: Prefix of a non-terminating execution; messages in transit shown adjacent to its receiver; message to be consumed next in the execution indicated by an arrow

In that execution, process p_1 has proposed 0 while the other two processes have proposed 1. In the first state in Fig. 3, each process has sent its proposal value to its successor. (A message in transit is shown adjacent to its receiver.) The arrow indicates the message that will be consumed next, resulting in the sending of a change message to p_3 (second state in Fig. 3). Now the change message is consumed, which results in the third state shown in the figure. The third state is symmetric to the first state (all values flipped and configuration rotated clockwise by 120 degrees). Now p_2 receives the next message and sends a change message to p_1 causing p_1 to change its estimate. After six such double steps, we

have reproduced the first state in Fig. 3. Repeating this ad infinitum, we obtain an infinite execution where each process takes infinitely many steps and no process ever decides.

4. CIRCUMVENTING THE IMPOSSIBILITY

In this section, we discuss various additional concepts that guarantee termination of our algorithm and therefore allow to solve the crash-tolerant consensus problem.

4.1 Initially dead processes

Fischer, Lynch, and Paterson showed already in their seminal paper [6] that not exactly the existence of crashed processes is the problem but rather the uncertainty when these crashes occur. They showed that the problem is solvable under the assumption that all processes that crash have crashed already initially. Such an assumption can be justified in systems where consensus is invoked rarely such that the likelihood that a process crashes during a consensus phase is negligible.

To achieve consensus under the assumption that all crashing processes are initially dead, we refine our algorithm by adding the following rule:

A process changes its estimate upon a change message only if that change message is the very first message it receives.

I.e., the extra condition \mathcal{D} in line 10 of Fig. 2 is instantiated by a boolean variable that is initially *true* and set to *false* after the first message is received.

That the algorithm indeed satisfies termination under the above assumption is established by a simple case analysis. First observe that the algorithm terminates when all proposal values are identical.

If there is one process that is initially dead, the other processes will reach a decision and terminate. Similarly, if there are two neighbors which mutually receive their messages first, the execution terminates. If that is not the case, then we have a process q that has the same proposal value as its predecessor p and either p receives q 's message first or q receives p 's message first. The former case cannot occur due to the definition of the algorithm and the latter case implies immediate termination.

4.2 Synchrony

A look at the non-terminating execution from Sect. 3.3 reveals that a decision is always prevented because a message from the predecessor arrives “too late”. A common approach to enforce termination is therefore to let a process wait some finite period of time for this message. Of course, a process cannot wait indefinitely because the predecessor may have crashed.

A system is *synchronous* if the duration of process steps and message transmission delays is bounded by some constant that is known to all processes. Let K be then an upper bound for the round-trip delay between two processes, i.e., the time that is needed from sending a message to a neighbor until receiving a reply to that message. If we instantiate \mathcal{D} for a single process, say p_1 , with the condition

More than K time units have elapsed since the most recent sending of a change message in line 9,

then the algorithm terminates.

To see why this is sufficient, assume that all processes are correct and p_1 sends a change message to its predecessor p_3 . Since the receipt of that message at p_3 is not delayed by a timeout, p_3 will reply with its changed estimate which is then received by p_1 . Because of synchrony, all this must happen within K time. The value sent by p_3 must necessarily be equal to p_1 's estimate, and since p_1 has blocked any message receipts from p_2 for K time, it will decide.

4.3 Partial synchrony

The system need not be synchronous in order to guarantee termination. It is sufficient that it is *partially synchronous*. Dwork et al. [5] introduced two forms of *partial synchrony*:

- Time bounds for process steps and communication delays exist, but they are not known.
- Bounds exist and are known, but they only hold eventually.

Chandra and Toueg [4] introduced a third form of partial synchrony:

- Bounds exist, but they are not known *and* they only hold eventually.

which is apparently the weakest common assumption of the former two forms. We instantiate \mathcal{D} now at one process with an *adaptive* timeout, i.e., we use the timeout mechanism from Sect. 4.2 where the timer is now initially set to some arbitrary value and that value is increased by some positive constant whenever the timer is set again. Eventually, the timer is set to a value that is larger than the actual bound of the round-trip delay and will not get smaller. Eventually, the system will become synchronous and the time bounds will hold, which in turn guarantee termination according to our discussion in Sect. 4.2.

4.4 Failure detectors

The timeout mechanism in Sect. 4.3 can be seen as a way to detect crashes, albeit in an unreliable way (reliable in case of synchrony). We can generalize this mechanism into a service offered by an additional failure detection module which runs in parallel to the consensus algorithm. In this module, every process regularly sends an “are you alive” message to its predecessor and starts a timeout of length K . Upon receiving such a message, a process immediately replies with an “I am alive” message. We say that a process *suspects* its predecessor if the timeout occurs before a message from that neighbor arrives. Suspicions may be incorrect meaning that it does not necessarily imply a crash under partial synchrony. However the failure detector guarantees the following two properties:

1. *Completeness*: Each crashed process will eventually be suspected by its successor and
2. *Eventual Accuracy*: There is a correct process that is eventually not suspected by its predecessor.

These two properties are also sufficient to guarantee termination of the algorithm, that is if \mathcal{D} is instantiated at each process with a query “Is my predecessor crashed?” to any failure detector with the two properties above then the algorithm terminates.

Chandra and Toueg [4] proposed the notion of an *unreliable failure detector* to abstract from the concrete implementation details of the detection and the properties of the model. They showed that a failure detector with the following two properties is sufficient to solve consensus if less than half of the processes may crash:

1. Each crashed process will eventually be suspected by some correct process.
2. There is a correct process that is eventually not suspected by any correct process.

A failure detector with these two properties is called *eventually weak*. The failure detector defined above satisfies the properties of an eventually weak failure detector and therefore allows to solve consensus. Viewing the timeout mechanism in terms of a failure detector can simplify the reasoning about the properties of the algorithm by encapsulating timing assumptions within the failure detector. Chandra, Hadzilacos, and Toueg [3] even showed that any failure detector that can be used to solve consensus can also be used to implement an eventually weak failure detector.

4.5 Randomization

Another look at the non-terminating execution from Fig. 3 shows that it is symmetric in repeating the same behavior over and over again. Another idea to reach a decision is therefore to break symmetry by using randomization. A process is free to choose a new value when it receives a change message. It then flips a coin to decide whether to keep its old estimate or to change it. (Function f in line 11 of the algorithm is instantiated with a coin flip with outcomes 0 and 1.) The new estimate, regardless whether it has changed or not, is sent to the successor. Recall that the agreement and validity properties are satisfied regardless of how f is chosen.

Whether this randomized algorithm terminates with probability 1 depends on the power of the *adversary* of the algorithm, that is a virtual entity that determines which of the messages in transit is received next in an execution. If the decision which message is received next can depend on the outcome of previous coin flips—we call the corresponding adversary *on-line adversary*—then termination with high probability is not guaranteed. That is, because the adversary can schedule a particular channel until the next configuration in the non-terminating execution in Fig. 3 is reached.

The on-line adversary uses its power to adjust the message delays dependent on the outcomes of the coin flips. Such a strong adversary is hard to come across in practice. It is often natural to assume that message delays and relative speeds of the processes do not depend on the outcome of the free choices of the processes. A corresponding adversary is called *off-line adversary*. An off-line adversary fixes the schedule in advance. Relative speeds and communication delays can still vary arbitrarily over time but they cannot be adjusted when the algorithm is running and the outcomes of coin flips become known.

Let a schedule be fixed, i.e., an order of channels that determines on which channel a message is delivered next. Given a schedule, we can define whether a particular outcome of a coin flip is *favorable* (for achieving termination): The outcome of a coin flip at process p is favorable if

1. it yields the current estimate of p 's successor when the channel between p and p 's successor is scheduled next,

2. and if it yields the current estimate of p 's predecessor otherwise.

A simple but tedious case analysis establishes that any three successive favorable coin flips guarantee termination. Therefore, the algorithm terminates with probability 1 under an off-line adversary. There are also randomized asynchronous consensus algorithms that terminate with probability 1 against an on-line adversary when less than half of the processes crash [2].

4.6 Input vector restrictions

Recently, Mostefaoui *et al.* introduced the *condition-based approach* to solve consensus [7]. The idea of this approach is to identify a restriction on the set of input vectors of the protocol (a *condition*) so that it terminates even in purely asynchronous systems and have an algorithm that maintains safety even if an input vector occurs which is not in that set. The motivation for this approach stems from the observation that not all input configurations are equally likely in practice. Identifying the conditions under which the protocol terminates thus gives an indication on how likely it will terminate in practice.

There is just one condition under which our protocol always terminates. It is the set containing the two input vectors (0, 0, 0) and (1, 1, 1). For all other input vectors, our algorithm at least guarantees validity and agreement.

5. CONCLUSION

In this exercise we showed how a simple algorithm can be used to illustrate the impossibility of crash-tolerant consensus in asynchronous distributed systems. We feel that this exercise can be used by students while approaching the problem for the first time. The presented algorithm achieves its simplicity by restricting itself to the case of three processes of which at most one may fail. This resilience is optimal for most of the concepts discussed here, so if there is only one crash to be tolerated this algorithm can be used. To tolerate more failures, we refer to the algorithms in the literature [2, 4–7]. However, the ability to withstand more failures comes at the price of unbounded states of processes, unbounded message sizes, and more complex communication schemes—at least with the known algorithms. While we have focussed on binary valued consensus in this text, there exist simple extensions for the non-binary case. We leave this case as an exercise to the reader.

Acknowledgement. We would like to thank Partha Dutta for comments, which improved the presentation of the material.

6. REFERENCES

- [1] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.
- [2] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.
- [3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

- [4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
- [5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.
- [6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.
- [7] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 153–162. ACM Press, 2001.
- [8] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.
- [9] H. Völzer. A constructive proof for FLP. *Information Processing Letters*, 92:83–87, 2004.