Chapter 17

Byzantine Agreement

In order to make flying safer, researchers studied possible failures of various sensors and machines used in airplanes. While trying to model the failures, they were confronted with the following problem: Failing machines did not just crash, instead they sometimes showed an unusual behavior before stopping completely. With these insights researchers proposed a more general failure model.

Definition 17.1 (Byzantine). A node which can have arbitrary behavior is called *byzantine*. This includes “anything imaginable”, e.g., not sending any messages at all, or sending different and wrong messages to different neighbors, or lying about the input value.

Remarks:
- Byzantine behavior also includes collusion, i.e., all byzantine nodes are being controlled by the same adversary.
- We assume that any two nodes communicate directly, and that no node can forge an incorrect sender address. This is a requirement, such that a single byzantine node cannot simply impersonate all nodes!
- We call non-byzantine nodes *correct* nodes.

Definition 17.2 (Byzantine Agreement). Finding consensus as in Definition 16.1 in a system with byzantine nodes is called *byzantine agreement*. An algorithm is \( f \)-resilient if it still works correctly with \( f \) byzantine nodes.

Remarks:
- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliver values in \( \mathbb{R} \), all-same validity is in most scenarios not really useful.

Definition 17.3 (Any-Input Validity). The decision value must be the input value of *any* node.

Remarks:
- This is the validity definition we used for consensus, in Definition 16.1.
- Does this definition still make sense in the presence of byzantine nodes? What if byzantine nodes lie about their inputs?
- We would wish for a validity definition that differentiates between byzantine and correct inputs.

Definition 17.4 (Correct-Input Validity). The decision value must be the input value of a *correct* node.

Remarks:
- Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node.
- Here is an alternative.

Definition 17.5 (All-Same Validity). If all correct nodes start with the same input \( v \), the decision value must be \( v \).

Remarks:
- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliver values in \( \mathbb{R} \), all-same validity is in most scenarios not really useful.

Definition 17.6 (Median Validity). If the input values are orderable, e.g. \( v \in \mathbb{R} \), byzantine outliers can be prevented by agreeing on a value close to the median of the correct input values – how close depends on the number of byzantine nodes \( f \).

Remarks:
- Is byzantine agreement possible? If yes, what validity condition?
- Let us try to find an algorithm which tolerates 1 single byzantine node, first restricting to the so-called synchronous model.

Model 17.7 (synchronous). In the *synchronous model*, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and do some local computation.

Definition 17.8 (synchronous runtime). For algorithms in the synchronous model, the *runtime* is simply the number of rounds from the start of the execution to its completion in the worst case (every legal input, every execution scenario).
17.2 How Many Byzantine Nodes?

Algorithm 17.9 Byzantine Agreement with $f = 1$.

1: Code for node $u$, with input value $x$.
2: Send tuple$(u, x)$ to all other nodes
3: Receive tuple$(v, y)$ from all other nodes $v$
4: Store all received tuple$(v, y)$ in a set $S_v$

Round 1
5: Send set $S_u$ to all other nodes
6: Receive sets $S_v$ from all nodes $v$
7: $T =$ set of tuple$(v, y)$ seen in at least two sets $S_v$, including own $S_u$
8: Let tuple$(v, y) \in T$ be the tuple with the smallest value $y$
9: Decide on value $y$

Remarks:
- Byzantine nodes may not follow the protocol and send syntactically incorrect messages. Such messages can easily be detected and discarded. It is worse if Byzantine nodes send syntactically correct messages, but with bogus content, e.g., they send different messages to different nodes.
- Some of these mistakes cannot easily be detected: For example, if a Byzantine node sends different values to different nodes in the first round; such values will be put into $S_u$. However, some mistakes can and must be detected: Observe that all nodes only relay information in Round 2, and do not repeat their own value. So, if a Byzantine node sends a set $S_v$, which contains a tuple$(v, y)$, this tuple must be removed by $u$ from $S_u$ upon receiving it (Line 6).
- Recall that we assumed that nodes cannot forge their source address; thus, if a node receives tuple$(v, y)$ in Round 1, it is guaranteed that this message was sent by $v$.

Lemma 17.10. If $n \geq 4$, all correct nodes have the same set $T$.

Proof. With $f = 1$ and $n \geq 4$, we have at least 3 correct nodes. A correct node will see every correct value at least twice, once directly from another correct node, and once through the third correct node. So all correct values are in $T$. If the Byzantine node sends the same value to at least 2 other (correct) nodes, all correct nodes will see the value twice, so all add it to set $T$. If the Byzantine node sends all different values to the correct nodes, none of these values will end up in any set $T$.

Theorem 17.11. Algorithm 17.9 reaches Byzantine agreement if $n \geq 4$.

Proof. We need to show agreement, any-input validity and termination. With Lemma 17.10 we know that all correct nodes have the same set $T$, and therefore agree on the same minimum value. The nodes agree on a value proposed by any node, so any-input validity holds. Moreover, the algorithm terminates after two rounds.

Remarks:
- If $n > 4$ the Byzantine node can put multiple values into $T$.
- Algorithm 17.9 only provides any-input agreement, which is question-able in the Byzantine context: Assume a Byzantine node sends different values to different nodes, what is its input value in that case?
- Algorithm 17.9 can be slightly modified to achieve all-same validity by choosing the smallest value that occurs at least twice.
- The idea of this algorithm can be generalized for any $f$ and $n > 3f$. In the generalization, every node sends in every of $f + 1$ rounds all information it learned so far to all other nodes. In other words, message size increases exponentially with $f$.
- Does Algorithm 17.9 also work with $n = 3$?

Theorem 17.12. Three nodes cannot reach Byzantine agreement with all-same validity if one node among them is Byzantine.

Proof. We will assume that the three nodes satisfy all-same validity and show that they will violate the agreement condition under this assumption.

In order to achieve all-same validity, nodes have to deterministically decide for a value $x$ if it is the input value of every correct node. Recall that a Byzantine node which follows the protocol is indistinguishable from a correct node. Assume a correct node sees that $n - f$ nodes including itself have an input value $x$. Then, by all-same validity, this correct node must deterministically decide for $x$.

In the case of three nodes ($n - f = 2$), a node has to decide on its own input value if another node has the same input value. Let us call the three nodes $u$, $v$, and $w$. If correct node $u$ has input 0 and correct node $v$ has input 1, the Byzantine node $w$ can fool them by telling $u$ that its value is 0 and simultaneously telling $v$ that its value is 1. By all-same validity, this leads to $u$ and $v$ deciding on two different values, which violates the agreement condition. Even if $u$ talks to $v$, and they figure out that they have different assumptions about $w$’s value, $u$ cannot distinguish whether $w$ or $v$ is Byzantine.

Theorem 17.13. A network with $n$ nodes cannot reach Byzantine agreement with $f \geq n/3$ Byzantine nodes.

Proof. Assume (for the sake of contradiction) that there exists an algorithm $A$ that reaches Byzantine agreement for $n$ nodes with $f \geq n/3$. Byzantine nodes. We will show that $A$ cannot satisfy all-same validity and agreement simultaneously.

Let us divide the $n$ nodes into three groups of size $n/3$ (either $[n/3]$ or $[n/3]$, if $n$ is not divisible by 3). Assume that one group of size $[n/3] \geq n/3$ contains only Byzantine and the other two groups only correct nodes. Let one group of correct nodes start with input value 0 and the other with input value 1. As in Lemma 17.12, the group of Byzantine nodes supports the input
value of each node, so each correct node observes at least \( n - f \) nodes who support its own input value. Because of all-same validity, every correct node has to deterministically decide on its own input value. Since the two groups of correct nodes had different input values, the nodes will decide on different values respectively, thus violating the agreement property.

17.3 The King Algorithm

Algorithm 17.14 King Algorithm (for \( f < n/3 \))

1. \( x = \) my input value
2. for phase = 1 to \( f + 1 \) do
   3. Broadcast value(\( x \))
   4. if some value(y) received at least \( n - f \) times then
   5. Broadcast propose(y)
   6. end if
   7. if some propose(z) received more than \( f \) times then
   8. \( x = z \)
   9. end if
10. Let node \( v_i \) be the predefined king of this phase \( i \)
11. The king \( v_i \) broadcasts its current value \( w \)
12. if received strictly less than \( n - f \) propose(y) then
13. \( x = w \)
14. end if
15. end for

Lemma 17.15. Algorithm 17.14 fulfills the all-same validity.

Proof. If all correct nodes start with the same value, all correct nodes propose it in Line 5. All correct nodes will receive at least \( n - f \) proposals, i.e., all correct nodes will stick with this value, and never change it to the king’s value. This holds for all phases.

Lemma 17.16. If a correct node proposes \( x \), no other correct node proposes \( y \), with \( y \neq x \), if \( n > 3f \).

Proof. Assume (for the sake of contradiction) that a correct node proposes value \( x \) and another correct node proposes value \( y \). Since a good node only proposes a value if it heard at least \( n - f \) value messages, we know that both nodes must have received their value from at least \( n - 2f \) distinct correct nodes (as at most \( f \) nodes can behave byzantine and send \( x \) to one node and \( y \) to the other one). Hence, there must be a total of at least \( 2(n - 2f) + f = 2n - 3f \) nodes in the system. Using \( 3f < n \), we have \( 2n - 3f > n \) nodes, a contradiction.

17.4 Lower Bound on Number of Rounds

Theorem 17.20. A synchronous algorithm solving consensus in the presence of \( f \) crashing nodes needs at least \( f + 1 \) rounds, if nodes decide for the maximum seen value.

Proof. Let us assume (for the sake of contradiction) that some algorithm A solves consensus in \( f \) rounds. Some node \( u_1 \) has the smallest input value \( x \), but in the first round \( u_1 \) can send its information (including information about its value \( x \)) to only some other node \( u_2 \) before \( u_1 \) crashes. Unfortunately, in the second round, the only witness \( u_2 \) of \( x \) also sends \( x \) to exactly one other node \( u_3 \) before \( u_2 \) crashes. This will be repeated, so in round \( f \) only node \( u_{f+1} \) knows about the smallest value \( x \). As the algorithm terminates in round \( f \), node \( u_{f+1} \) will decide on value \( x \), all other surviving (correct) nodes will decide on values larger than \( x \).
Remarks:

- A general proof without the restriction to decide for the minimum value exists as well.
- Since byzantine nodes can also just crash, this lower bound also holds for byzantine agreement, so Algorithm 17.14 has an asymptotically optimal runtime.
- So far all our byzantine agreement algorithms assume the synchronous model. Can byzantine agreement be solved in the asynchronous model?

17.5 Asynchronous Byzantine Agreement

Algorithm 17.21 Asynchronous Byzantine Agreement (Ben-Or, for \( f < \frac{n}{3} \))

1: \( x_u \in \{0, 1\} \) \hspace{1cm} \text{\(<\) input bit}
2: round = 1 \hspace{1cm} \text{\(<\) round}
3: while true do
4: Broadcast propose(\( x_{\text{round}} \))
5: Wait until \( n - f \) propose messages of current round arrived
6: if at least \( n/2 + 3f + 1 \) propose messages contain same value \( x \) then
7: Broadcast propose(\( x, \text{round} + 1 \))
8: Decide for \( x \) and terminate
9: else if at least \( n/2 + f + 1 \) propose messages contain same value \( x \) then
10: \( x_u = x \)
11: else
12: choose \( x_u \) randomly, with \( \Pr[x_u = 0] = \Pr[x_u = 1] = 1/2 \)
13: end if
14: round = round + 1
15: end while

Lemma 17.22. Let a correct node choose value \( x \) in Line 10, then no other correct node chooses value \( y \neq x \) in Line 10.

Proof. For the sake of contradiction, assume that both 0 and 1 are chosen in Line 10. This means that both 0 and 1 had been proposed by at least \( n/2 + 1 \) out of \( n - f \) correct nodes. In other words, we have a total of at least \( 2 \cdot n/2 + 2 = n + 2 > n - f \) correct nodes. Contradiction! \( \square \)

Theorem 17.23. Algorithm 17.21 solves binary byzantine agreement as in Definition 17.2 for up to \( f < \frac{n}{3} \) byzantine nodes.

Proof. First note that it is not a problem to wait for \( n - f \) propose messages in Line 5, since at most \( f \) nodes are byzantine. If all correct nodes have the same input value \( x \), then all (except the \( f \) byzantine nodes) will propose the same value \( x \). Thus, every node receives at least \( n - 2f \) propose messages containing \( x \). Observe that for \( f < \frac{n}{10} \), we get \( n - 2f > n/2 + 3f \) and the nodes will decide on \( x \) in the first round already. We have established all-same validity! If the correct nodes have different (binary) input values, the validity condition becomes trivial as any result is fine.

17.6 Random Oracle and Bitstring

Definition 17.24 (Random Oracle). A random oracle is a trusted (non-byzantine) random source which can generate random values.

Algorithm 17.25 Shared Coin with Magic Random Oracle

1: return \( c_i \), where \( c_i \) is \( i \)th random bit by oracle
Remarks:
• Algorithm 17.25, as well as the following shared coin algorithms, will for instance be called in Line 12 of Algorithm 17.21. So instead of every node throwing a local coin (and hoping that they all throw the same), the nodes throw a shared coin. In other words, the value $x_u$ in Line 12 of Algorithm 17.21 will be set to the return value of the shared coin subroutine.
• We have already seen a shared coin in Algorithm 16.21. This concept deserves a proper definition.

Definition 17.26 (Shared Coin). A shared coin is a binary random variable shared among all nodes. It is 0 for all nodes with constant probability and 1 for all nodes with constant probability. The shared coin is allowed to fail (be 0 for some nodes and  1 for other nodes) with constant probability.

Theorem 17.27. Algorithm 17.25 plugged into Algorithm 17.21 solves asynchronous byzantine agreement in expected constant number of rounds.

Proof. If there is a large majority for one of the input values in the system, all nodes will decide within two rounds since Algorithm 17.21 satisfies all-same-validity; the shared coin is not even used.

If there is no significant majority for any of the input values at the beginning of algorithm 17.21, all correct nodes will run Algorithm 17.25. Therefore, they will set their new value to the bit given by the random oracle and terminate in the following round.

If neither of the above cases holds, some of the nodes see an $n/2 + f + 1$ majority for one of the input values, while other nodes rely on the oracle. With probability $1/2$, the value of the oracle will coincide with the deterministic majority value of the other nodes. Therefore, with probability $1/2$, the nodes will terminate in the following round. The expected number of rounds for termination in this case is 3.

Remarks:
• Unfortunately, random oracles are a bit like pink fluffy unicorns: they do not really exist in the real world. Can we fix that?

Definition 17.28 (Random Bitstring). A random bitstring is a string of random binary values, known to all participating nodes when starting a protocol.

Algorithm 17.29 Naive Shared Coin with Random Bitstring

1: return $b_i$, where $b_i$ is $i$th bit in common random bitstring

Remarks:
• But is such a precomputed bitstring really random enough? We should be worried because of Theorem 16.13.

Theorem 17.30. If the scheduling is worst-case, Algorithm 17.29 plugged into Algorithm 17.21 does not terminate.

Proof. We start Algorithm 17.29 with the following input: $n/2 + f + 1$ nodes have input value 1, and $n/2 - f - 1$ nodes have input value 0. Assume w.l.o.g. that the first bit of the random bitstring is 0.

If the second random bit in the bitstring is also 0, then a worst-case scheduler will let $n/2 + f + 1$ nodes see all $n/2 + f + 1$ values 1, these will therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 - f - 1$ nodes receive strictly less than $n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not come to a decision in this round. Moreover, we have created the very same distribution of values for the next round (which has also random bit 0).

If the second random bit in the bitstring is 1, then a worst-case scheduler can let $n/2 - f - 1$ nodes see all $n/2 + f + 1$ values 1, and therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 + f + 1$ nodes receive strictly less than $n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not decide in this round. And we have created the symmetric situation for input value 1 that is coming in the next round.

So if the current and the next random bit are known, worst-case scheduling will keep the system in one of two symmetric states that never decide.

Remarks:
• Theorem 17.30 shows that a worst-case scheduler cannot be allowed to know the random bits of the future.
• Note that in the proof of Theorem 17.30 we did not even use any Byzantine nodes. Just bad scheduling was enough to prevent termination.
• Worst-case scheduling is an issue that we have not considered so far, in particular in Chapter 16 we implicitly assumed that message scheduling was random. What if scheduling is worst-case in Algorithm 16.21? A more general question is what if scheduling is done by a malicious scheduler.

Lemma 17.31. Algorithm 16.21 has exponential expected running time under worst-case scheduling.

Proof. In Algorithm 16.21, worst-case scheduling may hide up to $f$ rare zero co-blips. In order to receive a zero as the outcome of the shared coin, the nodes need to generate at least $f + 1$ zeros. The probability for this to happen is $(1/n)^{f+1}$, which is exponentially small for $f \in \Omega(n)$. In other words, with worst-case scheduling, with probability 1 − $(1/n)^{f+1}$ the shared coin will be 1. The worst-case scheduler must make sure that some nodes will always deterministically go for 0, and the algorithm needs $n^{f+1}$ rounds until it terminates.

Chapter Notes

The project which started the study of Byzantine failures was called SIFT and was founded by NASA [WLG+78] and the research regarding Byzantine agreement started to get significant attention with the results by Pease, Shostak, and...
Lamport [PSL80, LSP82]. In [PSL80] they presented the generalized version of Algorithm 17.9 and also showed that byzantine agreement is unachievable for \( n \leq 3f \). The algorithm presented in that paper is nowadays called Exponential Information Gathering (EIG), due to the exponential size of the messages.

There are many algorithms for the byzantine agreement problem. For example the Queen Algorithm [BG98] which has a better runtime than the King algorithm [BGP89], but tolerates less failures. That byzantine agreement requires at least \( f + 1 \) many rounds was shown by Dolev and Strong [DS83], based on a more complicated proof from Fischer and Lynch [FL82].

While many algorithms for the synchronous model have been around for a long time, the asynchronous model is a lot harder. The only results were by Ben-Or and Bracha. Ben-Or [Ben83] was able to tolerate \( f < n/5 \). Bracha [BT85] improved this tolerance to \( f < n/3 \).

Nearly all developed algorithms only satisfy all-same validity. There are a few exceptions, e.g., correct-input validity [FG03], available if the initial values are from a finite domain, median validity [SW15, MW18, DGM11] if the input values are orderable, or values inside the convex hull of all correct input values [VG13, MH13, MHVG15] if the input is multidimensional.

Before the term byzantine was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved – for obvious reasons – to the historic term byzantine [LSP82].

Hat tip to Peter Robinson for noting how to improve Algorithm 17.9 to all-same validity. This chapter was written in collaboration with Barbara Keller.

**Bibliography**


