Chapter 3

Consensus Fundamentals

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

Isaac Asimov

In this chapter, we will show how to solve consensus. The consensus problem is a fundamental research problem of computer science that led to various proposals over the last four decades. We will study some of these proposals and learn about their tolerance to failures and their complexities.

3.1 Introduction

Today, with the recent advent of blockchains, various consensus implementations were proposed to make replicas reach an agreement on the order of blocks of transactions updating the distributed ledger. However, consensus has been known to be unsolvable in the general model since 1985 [FLP85]. While existing protocols were designed these past fourty years to solve consensus under various assumptions, it remains unclear what are the guarantees offered by blockchain consensus algorithms and what are the necessary conditions for these guarantees to be satisfied. While the source code of most blockchain protocols is publicly available, the theoretical ramifications of the blockchain abstraction are rather informal. As main blockchain systems, like Bitcoin [Nak08] and Ethereum [Woo15], are now used to trade millions of US$ every day[1], it has become crucial to precisely identify its theoretical ramifications to anticipate the situations where large volume of assets could be lost.

3.2 Consensus without failures

To simplify the presentation, let us first assume that no nodes fail and that the communication is synchronous. This way, all nodes can proceed synchronously, all executing the same line of the algorithm in parallel before proceeding to the next line, as if they were doing it at the same time. Recall that assuming synchrony actually means that each node can upper-bound the delay Δ it takes to transmit a message, hence guaranteeing that after sending a message, waiting Δ time is enough to guarantee that the receiving node delivers the message.

3.2.1 Consensus algorithm without failures and with synchrony

Algorithm 5 depicts an algorithm that solves consensus without failures and with synchrony. More precisely, function propose (lines 2–6) invoked by all nodes, that will decide the same value. The execution of the distributed algorithm simply consists of every node p_i broadcasting its initial value v_i to the other nodes at line 3. As a result of this sending, each node p_i then receives messages from other nodes p_j where $j \neq i$ at line 4. Finally, p_i stores these newly learned values in V at line 5 before deciding the minimum of these values at line 6.

[1] https://coinmarketcap.com/exchanges/volume/24-hour/.

Algorithm 5 A consensus algorithm without failures and with synchrony at p_i

```
1: $V$, the set of known values, it is initially set to $\{v_i\}$ the singleton containing the value proposed
   by $p_i$

2: propose$(v_i)_i$:
3:     send($\{v \in V : p_i$ did not send $v$ yet$\}$)
4:     receive$(S_j)$ from all $p_j, j \neq i$
5:     $V = V \cup S_j$ for all $j$
6:     decide(min$(V)$)
```

3.2.2 Correctness of the consensus algorithm without failures

This algorithm solves the consensus problem because it solves the three properties of the consensus problem we saw in Chapter 3. In particular, executing the function will eventually lead to line 6 where the node decides, hence guaranteeing termination. All nodes will decide the same value, because their set V will contain the values of all nodes as they are sent by all and received by all synchronously, which guarantees agreement. Finally, the minimum value guarantees that the value decided is one of the values of V that have been proposed by the nodes. Note that we pick the minimum among all values of V at line 6 while the algorithm could use another deterministic picking function, returning for example the median of these values V.

3.2.3 Complexities of the consensus algorithm without failures

The message complexity measures the number of messages that are sent during the algorithm. The message complexity of Algorithm 5 is quadratic because it simply consists of nodes exchanging messages in an all-to-all fashion. Hence we have n nodes sending one message to each of the other $n-1$ nodes. This leads to $O(n^2)$ messages.

The communication complexity measures the number of bits that are sent during the algorithm. Note that the communication complexity differs from the message complexity when messages do not contain a constant number of bits. The message complexity of Algorithm 5 is also $O(b \cdot n^2)$ if we consider that the number of bits to encode each message is b.

The time complexity is the number of message delays necessary to terminate the algorithm. The time complexity is different from the message complexity when multiple messages are sent in parallel. As Algorithm 5 sends all messages in parallel, it only takes $O(1)$ message delays to terminate.

3.3 Consensus with crash failures

Consider now that there are f crash failures in that f nodes can fail by crashing, where $0 < f \leq n$. As before let assume that the communication is synchronous so that all nodes execute the same line of the algorithm before proceeding to the next line. The algorithm consists of executing a loop at lines 4–7 of $f + 1$ iterations, in each of which the nodes behave similarly to Aglorithm 5: they exchange at lines 5 and 6 the values that they have not exchanged yet and add to the set V all the values they know as their input value or as the values they received from other nodes at line 7. At the end of the loop, the node extract the minimum value among the obtained set V and decides this minimum value at line 8.

Algorithm 6 A consensus algorithm with crash failures and with synchrony at p_i

```
1: $V$, the set of known values, it is initially set to $\{v_i\}$ the singleton containing the value proposed
   by $p_i$
2: $f$, the maximum number of nodes that can fail

3: propose$(v_i)_i$:
4:   for $k = 1$ to $f + 1$ do
5:     send($\{v \in V : p_i$ did not send $v$ yet$\}$)
6:     receive$(S_j)$ from all $p_j, j \neq i$
7:     $V = V \cup S_j$ for all $j$
8:   decide$(\min(V))$
```

3.3.1 Correctness of the consensus algorithm with crash failures

In order to guarantee that Algorithm 6 solves consensus in the presence of f crash failures, we need to guarantee each of the three properties of the consensus problem: termination, agreement and validity.

The protocol terminates by deciding after the bounded number $f + 1$ of iterations of the loop. Note that if $f = 1$ the algorithm resembles Algorithm 5, in which case it terminates after only one iteration.

Agreement is reached because at the end of the loop, every pair of nodes p_i and p_j have the same sets of values $V_i = V_j$ for any i and j, hence the minimum of these sets that are decided are identical. We know $V_i = V_j$ because no nodes can send a different value from the value v_i it initially had or from the values it received from others. In particular, no nodes can send different values to different nodes because they either fail by crashing in which case they do not send anything or they are correct. The worst thing a node that fails by crashing can do in this algorithm is to send a value to some nodes but to fail at line 5 before sending to all, hence some nodes receive its value and some nodes do not receive it. In this case, however, we are guaranteed to have an iteration of the loop where no new failure happens as the number of iterations is strictly larger than the number of failures f. In this particular iteration, all values partially sent by faulty nodes are re-exchanged so that all nodes receive them.

Validity is guaranteed because this set V_i contains only proposed values for any node p_i. Note that the algorithm also works when f is as large as n. In this case, it is easy to see that validity and termination are guaranteed for the same reasons as above. The agreement is also guaranteed because there is no two correct nodes that decide differently as there are no correct nodes.

3.3.2 Complexity of the consensus algorithm with crash failures

The message complexity is $O(n^2)$ in each iteration of the loop because each node sends messages to all other other $n-1$ nodes. As there are $f+1$ iterations of this loop, we obtain $O((f+1)n^2)$ message complexity for the whole algorithm. Note that as f can be as large as n for this algorithm, this complexity is actually $O(n^3)$.

The communication complexity depends on the number of bits exchanged per message. Let b be the number of bits to encode the initial value of each node. As nodes exchange the values that they have not send yet, the messages can contain up to $(n-1)b$ bits. As we have $O((f+1)n^2)$ messages as mentioned above and each message is of size $O(bn)$, we obtain a communication complexity of $O(b(f+1)n^3)$ bits.

The time complexity follows from the number of iterations. There are $f+1$ iterations in the loop of this algorithm and in each of these iteration, messages can be exchanged in parallel. This leads to a time complexity of $O(f+1)$ message delays.

3.4 Problem of consensus with Byzantine failures

The previous definition of the consensus problem (Definition 1) is not well suited for the case where there are Byzantine failures. In particular, its validity property simply requires that the value decided is one of the proposed value, yet by definition a node experiencing a Byzantine failure can propose a value that is not desirable. Intuitively, we would like to avoid the consensus to output a value proposed exclusively by Byzantine nodes.

If we consider the *binary* Byzantine consensus problem, where the proposed and decided values are binary, i.e., they can either be 0 or 1, then one can change the validity property to require that the value decided is one of the values proposed by correct nodes. In this case, this new binary Byzantine consensus problem definition that inherits from the same agreement and termination properties as Definition 1, and replaces the validity property of Definition 1 by "the decided value is a value proposed by a correct node" could intuitively accept some solution. To illustrate what could be such a solution consider that $f < n/3$. To satisfy this new validity, it is sufficient to decide the value that was proposed $f+1$ times. As there are f Byzantine processes, no values proposed exclusively by Byzantine processes can be proposed by $f+1$ distinct processes. Note that this value is guaranteed to exist because there are at least $2f+1$ correct nodes and each correct node can only propose one of two values, either 0 or 1. Definition 5 will introduce an even more general definition of the binary Byzantine consensus problem.

If we consider the *multivalue* Byzantine consensus problem, where there could be more than two distinct values that are proposed and decided, then there cannot be a solution that solves the agreement and termination properties of Definition 1 as well as the validity stating "the decided value is a value proposed by a correct node". With multiple values, there is no guarantee that there will be one proposed value that predominates sufficiently to be detected by any correct process. Typically, if there is no value proposed by more than f distinct nodes, then it is unclear which value is proposed exclusively by Byzantine nodes and which value is proposed by some correct process. We thus adopt a slightly stronger notion of validity than the one of Definition 1 by also requiring that in the case where all nodes are correct and they all propose the same value, then we have to decide this value. As the following definition of Byzantine Consensus (BC) finds solutions when there are more than two distinct values, it also finds solutions when the values are binary.

Definition 3 (Byzantine Consensus) *Consider that each correct node proposes a value, the Byzantine consensus (BC) problem is for each of them to decide on a value in such a way that the following properties are satisfied:*

- *BC-Validity: Any decided value is a value proposed by some node and if all nodes are correct and they all propose the same value, then the correct nodes decide this value.*
- *Agreement: No two correct nodes decide differently.*
- *Termination: Every correct node eventually decides.*

3.5 Byzantine fault tolerance consensus

Let see how to solve the Byzantine consensus problem in order to cope with malicious behaviors that can be common in blockchain systems. To this end, we consider a simple model, where communication is synchronous in that every message is delivered within a known bounded time and that f is lower than $n/3$.

3.5.1 The EIG algorithm

We present the Exponential Information Gathering (EIG) algorithm. To understand the algorithm, each process maintains a tree structure, called an *EIG tree*, that has a degree n and a depth $f+2$ spanning level 0, at the root, to level $f+1$ at the leaf nodes. Initially, each process decorates its tree with the value it receives from other processes. If the value is not well formed, then the process uses label $\perp$ to indicate that the value is undefined. Otherwise it decorates the tree of the process with a label denoted $i_1, ..., i_k$ with value v to indicate that i_k

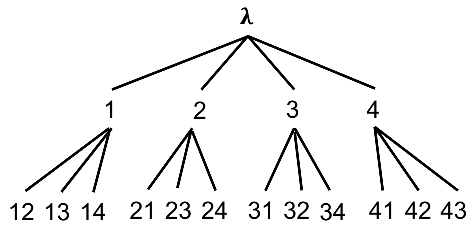


Figure 3.1: The labels of the EIG tree at each process when $n = 4$.

told i at round k that i_{k-1} told i_k at round $k-1$ that i_{k-2} told i_{k-1} at round $k-2$, ..., that i_1 told i_2 at round 1 that i_1's proposal is v.

For example, if $n = 4$ and $f = 1$, each process will build a tree whose labelling is depicted in Figure 3.1. If the node labelled 23 in the tree of p_1 (or process 1) is decorated with value 0 as it is depicted in Figure 3.2(b), then it indicates that process 3 informed process 1 that process 2 told him that its input value is 0.

At the end of the algorithm of $f+1$ rounds, p_i updates the tree by traversing it bottom-up. It decorates each node with an additional *newval* as follows:

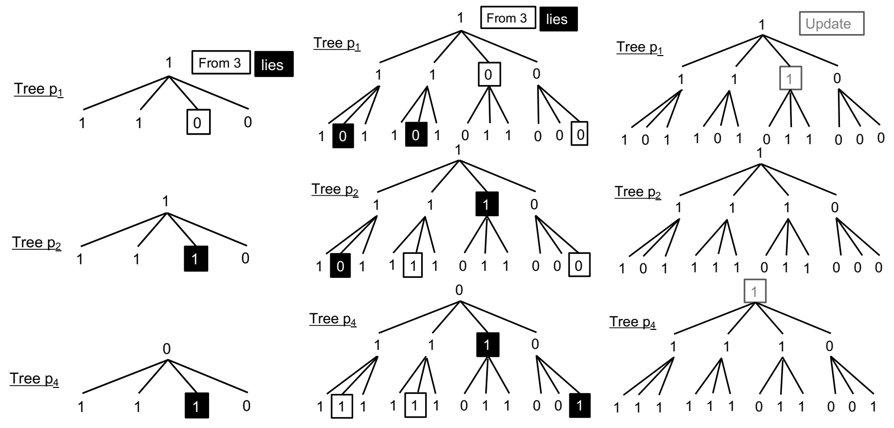
- For each leaf labeled x, $\mathit{newval}(x) = \mathit{val}(x)$;
- For each non-leaf node labeled x, $\mathit{newval}(x)$ is the *newval* held by a strict majority of the children of node x (or null if no such majority exists).

At the end, each process decides the value $\mathit{newval}(\lambda)$, which is the updated value at the root of the tree once the traversal completes.

3.5.2 Example with $n = 4$ and $f = 1$

To illustrate the algorithm, consider Figure 3.2 where $n = 4$ processes, namely p_1, p_2, p_3 and p_4, decorate their EIG tree as they exchange messages. We consider that we have $f = 1 < n/3$ processes that are Byzantine. Let p_3 be the Byzantine process and let 1, 1, 0 and 0 be the input values of p_1, p_2, p_3 and p_4, respectively. We are thus interested in the tree of processes p_1, p_2 and p_4 and how these correct processes reach a consensus despite p_3 lying to them.

All processes proceed in $f+1$ synchronous rounds where they exchange messages. After the first exchange, p_1, p_2, p_4 received 1, 1, 0 from p_1, p_2, p_4, respectively. As p_3 is Byzantine, it sent 0 to p_1, and lied about its value by sending 1 to both p_2 and p_4, even though its input value is 0. As a result, p_1, p_2, p_4 obtained the tree depicted in Figure 3.2(a). At the end of the following round, correct processes p_1, p_2 and p_4 exchange what they received previously. The Byzantine process p_3 pretends that it received 0 from p_1 and from p_2 to process p_1 as indicated in the bottom left subtree of the p_1's tree, at the top of Figure 3.2(b). The Byzantine process p_3 also lies to p_2 by saying that it received



(a) During one communication round, given that p_3 is Byzantine, p_1, p_2 and p_4 receives $\langle 1,1,0,0\rangle$, $\langle 1,1,1,0\rangle$, $\langle 1,1,1,0\rangle$ from p_1, p_2, p_3 and p_4, respectively. We can see that p_3 lies to both p_2 and p_4.

(b) During a second communication round, p_3 lies to p_1 and p_2 by saying that it received 0 from p_1 and lies to p_4 by saying that it received 1 from p_4.

(c) At the end, each process collects a *newval* at the root of each subtree, by traversing from the bottom to the top of the tree, in order to decide the value *newval*(λ) at the root of the tree.

Figure 3.2: The EIG algorithm consists of all correct processes, say p_1, p_2 and p_4, building a tree such that the node 23 in the tree of process 1 is labelled with the value that process 3 told process 1 that it received from process 2. In Fig. 3.2(a) and 3.2(b), values from process p_3 are depicted within a rectangle and its lies are within black rectangles.. In Fig. 3.2(c), the updates are framed in grey.

0 from p_1 as indicated in p_2's tree in the middle of Figure 3.2(b) and to p_4 saying that it received 1 from p_4.

Finally, during the bottom-up traversal, each process identifies the predominant values in the leaves of each subtree and sets the root of the corresponding subtree to this value. For example, in Figure 3.2(c), p_1's subtrees have all a majority of value 1 at their leaves, except the right-most subtree that has three 0 at its leaves, hence all subtrees have root value 1 except the right-most subtree that has root value 0. At the end they have all set their root value $\mathit{newval}(\lambda)$ to 1 and decide this value as the outcome of the consensus.

3.5.3 Complexity of the EIG algorithm

To decorate the tree, one has to go from one level of the EIG to the next by exchanging messages. As the decoration requires to go from top to bottom, there are $f+1$ exchanges to go through. As each exchange involves $O(n^2)$ messages, the message complexity becomes $O((f+1)n^2)$ or simply $O(fn^2)$.

There are $f+1$ rounds, in each of these rounds, messages can be exchanged in parallel. This leads to a time complexity of $O(f+1)$ message delays.

The communication complexity is exponential in the number of failures because the content of previous messages is piggybacked in new messages, which leads to a communication complexity of $O(bn^{f+1})$ bits if each value is of size b bits.

3.6 Conclusion

The problem of consensus, although seemingly simple, is not easy to solve when the processes or participants can fail. In particular, we have seen three solutions offering different fault tolerance. The first algorithm is simple but solves consensus when there are no failures. The second algorithm is a variant that solves consensus when participants may fail by crashing but it takes longer to execute than the first algorithm. The third algorithm solves consensus when $f < n/3$ processes among n can fail arbitrarily, however, it requires to exchange a lot more information as its communication complexity is exponential. In all these solutions we assume that the communication was synchronous, we explain in Chapter 4 why this can lead to dramatic consequences when executing these algorithms over the Internet.

3.7 Bibliographic notes

Designing Byzantine fault tolerant consensus algorithms is notoriously difficult. They are often large monolithic software [CL02], that are partially implemented [AGK$^+$15a], and their complexity makes them subject to flaws [TG19].

It is interesting to note that some blockchain technologies do not tolerate Byzantine failures [BCGH16, ABB$^+$18]. This is the case of Hyperledger Fabric [ABB+18] or R3 Corda [BCGH16] that make the assumption that all the permissionned nodes that participate in the consensus trust each other. Although the version 0.6 of Hyperledger Fabric was originally designed to cope with arbitrary failures, it has never been completed to go to production and a prototype Byzantine fault tolerant component [SBV18] does not make Hyperledger Fabric Byzantine fault tolerant.

Bibliography

[ABB$^+$18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, 2018.

[AGK$^+$15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[BCGH16] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction, 2016.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[Nak08] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008. http://www.bitcoin.org.

[SBV18] Joao Sousa, Alysson Bessani, and Marko Vukolić. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

[TG19] Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain Byzantine fault tolerance. In *6th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA'19)*, Oct 2019.

[Woo15] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015. Yellow paper.