Byzantine Fault Tolerance (BFT) Consensus

Blockchain @ uninsubria

Roberto Vicario

2024/2025

Contents

1 Byzantine Fault Tolerance (BFT) Problem

In a distributed network, not every participant can be trusted. Some may fail, lie, or act maliciously. This leads to the *Byzantine Fault Tolerance Problem*:

> **Problem:** *If all the generals coordinate their attack and strike at the same time, the battle is won. However, if two generals falsely claim they will attack but instead retreat, the battle is lost. This failure in coordination undermines the entire strategy.*

1.1 Fault Tolerance

Fault tolerance is the ability of a system to continue operating correctly even when some of its components fail:

- **Crash Faults:** Nodes may crash or become unresponsive.
- **Byzantine Faults:** Nodes may behave arbitrarily, including sending conflicting information to different nodes.

> **Theorem:** Let n be the total number of nodes in a distributed system and f be the maximum number of faulty nodes. To guarantee that honest nodes can reach consensus despite the presence of these faulty nodes, it is necessary that:

$$f \leq \frac{n-1}{3} \tag{1}$$

2 Byzantine Broadcast (BB)

> ***Synchronous***

This method allows a source node to broadcast a message to all other nodes in the network, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** The source node sends the message to all nodes.

2. **Propagation:** Each node forwards the message received from the source node to all other nodes.
3. **Acknowledgment:** Each node applies a decision rule (e.g., majority voting) based on the messages received to agree on the final value.
4. **Termination:** The protocol terminates when all honest nodes agree on the same value.

Single Point of Failure: This protocol relies on a single source node to initiate the broadcast. If this node is compromised, it can send conflicting messages to different nodes, potentially disrupting consensus and causing the entire system to never reach consensus.

2.1 Dolev-Strong Protocol

> ***Synchronous***

Dolev-Strong protocol allows a source node to broadcast a message to all other nodes in the network, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** The sender sends a signed message m to each node:

$$M_0 = (m, \sigma_S(m))$$

2. **Propagation:** For rounds $r = 1, \dots, f$, each node forwards the message to all other nodes, including its own signature i:

$$M_{r-1} = (m, \sigma_{i_0}(m), \sigma_{i_1}(m), \dots, \sigma_{i_{r-1}}(m))$$

3. **Acknowledgment:** Each node collects all received messages M_r. Messages with invalid values in the chain are discarded.
4. **Termination:** The consensus is reached when nodes reach the quorum of $2f+1$ messages. The final value is the one with the highest number of votes.

> **FLM (Fischer-Lynch-Merritt) Impossibility Result:** In synchronous systems, the absence of a *Public Key Infrastructure (PKI)*, makes impossible to achieve consensus.

3 Byzantine Agreement (BA)

> ***Asynchronous***

This method does not rely on a single source node to initiate the broadcast. Instead, it allows all nodes to participate in the consensus process, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** Each node starts with an initial private value.
2. **Agreement:** All honest nodes decide the same value.
3. **Validity:** If all honest nodes start with the same value, they will all decide on that value.
4. **Termination:** The protocol terminates when all honest nodes agree on the same value.

3.1 FLP (Fischer-Lynch-Patterson) Impossibility Result

> ***Asynchronous***

In asynchronous systems, does not exist a deterministic algorithm that can guarantee consensus in the presence of even a single faulty node. This could lead to a situation where the system never reaches consensus, as nodes may wait indefinitely for messages from other nodes.

$\boxed{PROOF}$

Let each node hold a private input value of either 0 or 1. Define a family of initial configurations X_i, where the first i nodes have value 0 and the remaining $n - i$ nodes have value 1:

$$X_i = (\underbrace{0, 0, \dots, 0}_{i}, \underbrace{1, 1, \dots, 1}_{n-i})$$

Since $X_0 = (0, 0, \dots, 0)$, by *Validity*, the output must be 0. Thus, X_0 is a 0-configuration. Similarly, $X_n = (1, 1, \dots, 1)$ must lead to output 1, and is a 1-configuration.

If we gradually flip the input bits from 0 to 1, moving from X_0 to X_n, there must exist a first index i such that:

- X_{i-1} is still a 0-configuration, but X_i is not a 0-configuration, and is not yet a 1-configuration either.

> **Lemma 1:** There exists at least one *Ambiguous Configuration*, where the output is not determined and *Termination* is not guaranteed.

Let C be an ambiguous configuration, one from which both decision values 0 and 1 are still possible through different executions. Then, there exists at least one event e (e.g., the delivery of a message to a process) such that executing e from C leads to another ambiguous configuration.

> **Lemma 2:** Even if an event is applied to an ambiguous configuration, it does not necessarily break the bivalence: the system can remain in a state where both outcomes (0 or 1) are still possible. This prevents the system from "forcing" a definitive decision in a finite number of steps.

Using Lemma 1 (existence of an ambiguous configuration) and Lemma 2 (persistence of ambiguity), one can construct an infinite, fair execution in which the system never reaches a decision, violating the *Termination* property of consensus.

Conclusion: Therefore, deterministic consensus is impossible in an asynchronous system with even one possible faulty node.

4 Rotating Leaders

> ***Partially Synchronous***

This method proposes a solution for *Single Point of Failure* problems by allowing nodes to take turns being the leader, ensuring that no single node has control over the entire system.

$\boxed{ALGORITHM}$

1. **Initialization:** A leader is selected.
2. **Proposal:** The leader proposes a value to the other nodes.
3. **Voting:** Each node votes on the proposed value, and if a majority agrees, the value is committed.
4. **Rotation:** After a round, the leadership role rotates to the next node.

4.1 DLS (Dwork-Lynch-Stockmeyer) Impossibility Result

> ***Partially Synchronous***

In a partially synchronous system, where message delivery time becomes bounded only after some unknown *Global Stabilization Time (GST)*, no deterministic algorithm can guarantee both safety and liveness in the presence of even one faulty process.

$\boxed{PROOF}$

1. Let N_2 be the faulty process, with N_1 and N_3 honest. Suppose N_1 has input 1 and N_3 has input 0.
2. The faulty process N_2 behaves inconsistently, sending messages to N_1 claiming its input is 1, and to N_3 claiming its input is 0.
3. Meanwhile, the adversary delays all communication between N_1 and N_3 until after both have made their decisions.
4. Because GST is unknown and message delays before GST can be unbounded, N_1 and N_3 cannot distinguish between delayed messages and crashed processes. Consequently, N_1, believing both it and N_2 are honest, may decide on 1; similarly, N_3, also trusting N_2, may decide on 0. This leads to:

$$v_1 \neq v_3$$

Conclusion: Therefore, no deterministic algorithm can achieve consensus in a partially synchronous system with even one faulty process.

> **CAP Principle:** No distributed system can simultaneously guarantee *Consistency*, *Availability*, and *Partition Tolerance*.

4.2 Tendermint Protocol

> ***Partially Synchronous***

Tendermint is a deterministic consensus protocol designed to work in the partially synchronous model, tolerating faulty nodes giving up safety during asynchronous periods.

$\boxed{ALGORITHM}$

1. **Propose:** A designated proposer (deterministically selected from the validator set) broadcasts a proposed block B_h for height h to all validators.
2. **Prevote:** Upon receiving a valid proposal B_h, each validator v_i broadcasts a prevote message:

$$\text{prevote}_i(h, r) = \begin{cases} \text{hash}(B_h), & \text{if } B_h \text{ is valid and received} \\ \text{none}, & \text{otherwise} \end{cases}$$

   where r is the current round number.

3. **Precommit:** If a validator observes prevotes for the same block hash from at least $\frac{2n}{3}$ validators, it broadcasts a precommit message:

$$\text{precommit}_i(h, r) = \begin{cases} \text{hash}(B_h), & \text{if prevotes}(h, r) \geq \frac{2n}{3} \text{ for } B_h \\ \text{none}, & \text{otherwise} \end{cases}$$

4. **Commit:** If a validator observes precommits for the same block hash from at least $\frac{2n}{3}$ validators, it commits B_h as the block at height h, finalizes the round, and proceeds to height $h + 1$.