WIKIPEDIA
The Free Encyclopedia

# Consensus (computer science)

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach **consensus**, or agree on some data value that is needed during computation. Example applications of consensus include agreeing on what transactions to commit to a database in which order, state machine replication, and atomic broadcasts. Real-world applications often requiring consensus include cloud computing, clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain, and others.

## Problem description

The consensus problem requires agreement among a number of processes (or agents) on a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault-tolerant or resilient. The processes must put forth their candidate values, communicate with one another, and agree on a single consensus value.

The consensus problem is a fundamental problem in controlling multi-agent systems. One approach to generating consensus is for all processes (agents) to agree on a majority value. In this context, a majority requires at least one more than half of the available votes (where each process is given a vote). However, one or more faulty processes may skew the resultant outcome such that consensus may not be reached or may be reached incorrectly.

Protocols that solve consensus problems are designed to deal with a limited number of faulty processes. These protocols must satisfy several requirements to be useful. For instance, a trivial protocol could have all processes output binary value 1. This is not useful; thus, the requirement is modified such that the production must depend on the input. That is, the output value of a consensus protocol must be the input value of some process. Another requirement is that a process may decide upon an output value only once, and this decision is irrevocable. A method is correct in an execution if it does not experience a failure. A consensus protocol tolerating halting failures must satisfy the following properties.[1]

**Termination**
: Eventually, every correct process decides some value.

**Integrity**
: If all the correct processes proposed the same value $v$, then any correct process must decide $v$.

**Agreement**
: Every correct process must agree on the same value.

Variations on the definition of *integrity* may be appropriate, according to the application. For example, a weaker type of integrity would be for the decision value to equal a value that some correct process proposed – not necessarily all of them.[1] There is also a condition known as **validity** in the literature which refers to the property that a message sent by a process must be delivered.[1]

A protocol that can correctly guarantee consensus amongst n processes of which at most t fail is said to be *t-resilient*.

In evaluating the performance of consensus protocols two factors of interest are *running time* and *message complexity*. Running time is given in Big O notation in the number of rounds of message exchange as a function of some input parameters (typically the number of processes and/or the size of the input domain). Message complexity refers to the amount of message traffic that is generated by the protocol. Other factors may include memory usage and the size of messages.

## Models of computation

Varying models of computation may define a "consensus problem". Some models may deal with fully connected graphs, while others may deal with rings and trees. In some models message authentication is allowed, whereas in others processes are completely anonymous. Shared memory models in which processes communicate by accessing objects in shared memory are also an important area of research.

### Communication channels with direct or transferable authentication

In most models of communication protocol participants communicate through *authenticated channels*. This means that messages are not anonymous, and receivers know the source of every message they receive. Some models assume a stronger, *transferable* form of authentication, where each *message* is signed by the sender, so that a receiver knows not just the immediate source of every message, but the participant that initially created the message. This stronger type of authentication is achieved by digital signatures, and when this stronger form of authentication is available, protocols can tolerate a larger number of faults.[2]

The two different authentication models are often called *oral communication* and *written communication* models. In an oral communication model, the immediate source of information is known, whereas in stronger, written communication models, every step along the receiver learns not just the immediate source of the message, but the communication history of the message.[3]

### Inputs and outputs of consensus

In the most traditional **single-value** consensus protocols such as Paxos, cooperating nodes agree on a single value such as an integer, which may be of variable size so as to encode useful metadata such as a transaction committed to a database.

A special case of the single-value consensus problem, called **binary consensus**, restricts the input, and hence the output domain, to a single binary digit {0,1}. While not highly useful by themselves, binary consensus protocols are often useful as building blocks in more general consensus protocols, especially for asynchronous consensus.

In **multi-valued** consensus protocols such as Multi-Paxos and Raft, the goal is to agree on not just a single value but a series of values over time, forming a progressively-growing history. While multi-valued consensus may be achieved naively by running multiple iterations of a single-valued consensus protocol in succession, many optimizations and other considerations such as reconfiguration support can make multi-valued consensus protocols more efficient in practice.

### Crash and Byzantine failures

There are two types of failures a process may undergo, a crash failure or a Byzantine failure. A *crash failure* occurs when a process abruptly stops and does not resume. *Byzantine failure*s are failures in which absolutely no conditions are imposed. For example, they may occur as a result of the malicious actions of an adversary. A process that experiences a Byzantine failure may send contradictory or conflicting data to other processes, or it may sleep and then resume activity after a lengthy delay. Of the two types of failures, Byzantine failures are far more disruptive.

Thus, a consensus protocol tolerating Byzantine failures must be resilient to every possible error that can occur.

A stronger version of consensus tolerating Byzantine failures is given by strengthening the Integrity constraint:

**Integrity**
: If a correct process decides $v$, then $v$ must have been proposed by some correct process.

## Asynchronous and synchronous systems

The consensus problem may be considered in the case of asynchronous or synchronous systems. While real world communications are often inherently asynchronous, it is more practical and often easier to model synchronous systems,[4] given that asynchronous systems naturally involve more issues than synchronous ones.

In synchronous systems, it is assumed that all communications proceed in *rounds*. In one round, a process may send all the messages it requires, while receiving all messages from other processes. In this manner, no message from one round may influence any messages sent within the same round.

### The FLP impossibility result for asynchronous deterministic consensus

In a fully asynchronous message-passing distributed system, in which at least one process may have a *crash failure*, it has been proven in the famous 1985 **FLP impossibility result** by Fischer, Lynch and Paterson that a deterministic algorithm for achieving consensus is impossible.[5] This impossibility result derives from worst-case scheduling scenarios, which are unlikely to occur in practice except in adversarial situations such as an intelligent denial-of-service attacker in the network. In most normal situations, process scheduling has a degree of natural randomness.[4]

In an asynchronous model, some forms of failures can be handled by a synchronous consensus protocol. For instance, the loss of a communication link may be modeled as a process which has suffered a Byzantine failure.

Randomized consensus algorithms can circumvent the FLP impossibility result by achieving both safety and liveness with overwhelming probability, even under worst-case scheduling scenarios such as an intelligent denial-of-service attacker in the network.[6]

## Permissioned versus permissionless consensus

Consensus algorithms traditionally assume that the set of participating nodes is fixed and given at the outset: that is, that some prior (manual or automatic) configuration process has **permissioned** a particular known group of participants who can authenticate each other as members of the group. In the absence of such a well-defined, closed group with authenticated members, a Sybil attack against an open consensus group can defeat even a Byzantine consensus algorithm, simply by creating enough virtual participants to overwhelm the fault tolerance threshold.

A **permissionless** consensus protocol, in contrast, allows anyone in the network to join dynamically and participate without prior permission, but instead imposes a different form of artificial cost or barrier to entry to mitigate the Sybil attack threat. Bitcoin introduced the first permissionless consensus protocol using proof of work and a difficulty adjustment function, in which participants compete to solve cryptographic hash puzzles, and probabilistically earn the right to commit blocks and earn associated rewards in proportion to their invested computational effort. Motivated in part by the high energy cost of this approach, subsequent permissionless consensus protocols have proposed or adopted other alternative participation rules for Sybil attack protection, such as proof of stake, proof of space, and proof of authority.

## Equivalency of agreement problems

Three agreement problems of interest are as follows.

### Terminating Reliable Broadcast

A collection of $n$ processes, numbered from $0$ to $n-1$, communicate by sending messages to one another. Process $0$ must transmit a value $v$ to all processes such that:

1. if process $0$ is correct, then every correct process receives $v$
2. for any two correct processes, each process receives the same value.

It is also known as The General's Problem.

### Consensus

Formal requirements for a consensus protocol may include:

- *Agreement*: All correct processes must agree on the same value.
- *Weak validity*: For each correct process, its output must be the input of some correct process.
- *Strong validity*: If all correct processes receive the same input value, then they must all output that value.
- *Termination*: All processes must eventually decide on an output value

### Weak Interactive Consistency

For $n$ processes in a partially synchronous system (the system alternates between good and bad periods of synchrony), each process chooses a private value. The processes communicate with each other by rounds to determine a public value and generate a consensus vector with the following requirements:[7]

1. if a correct process sends $v$, then all correct processes receive either $v$ or nothing (integrity property)
2. all messages sent in a round by a correct process are received in the same round by all correct processes (consistency property).

It can be shown that variations of these problems are equivalent in that the solution for a problem in one type of model may be the solution for another problem in another type of model. For example, a solution to the Weak Byzantine General problem in a synchronous authenticated message passing model leads to a solution for Weak Interactive Consistency.[8] An interactive consistency algorithm can solve the consensus problem by having each process choose the majority value in its consensus vector as its consensus value.[9]

## Solvability results for some agreement problems

There is a t-resilient anonymous synchronous protocol which solves the Byzantine Generals problem,[10][11] if $\frac{t}{n} < \frac{1}{3}$ and the Weak Byzantine Generals case[8] where $t$ is the number of failures and $n$ is the number of processes.

For systems with $n$ processors, of which $f$ are Byzantine, it has been shown that there exists no algorithm that solves the consensus problem for $n \leq 3f$ in the *oral-messages model*.[12] The proof is constructed by first showing the impossibility for the three-node case $n = 3$ and using this result to argue about partitions of processors. In the *written-messages model* there are protocols that can tolerate $n = f + 1$.[2]

In a fully asynchronous system there is no consensus solution that can tolerate one or more crash failures even when only requiring the non triviality property.[5] This result is sometimes called the FLP impossibility proof named after the authors Michael J. Fischer, Nancy Lynch, and Mike Paterson who were awarded a Dijkstra Prize for this significant work. The FLP result has been mechanically verified to hold even under fairness assumptions.[13] However, FLP does not state that consensus can never be reached: merely that under the model's assumptions, no algorithm can always reach consensus in bounded time. In practice it is highly unlikely to occur.

## Some consensus protocols

The Paxos consensus algorithm by Leslie Lamport, and variants of it such as Raft, are used pervasively in widely deployed distributed and cloud computing systems. These algorithms are typically synchronous, dependent on an elected leader to make progress, and tolerate only crashes and not Byzantine failures.

An example of a polynomial time binary consensus protocol that tolerates Byzantine failures is the Phase King algorithm by Garay and Berman.[14] The algorithm solves consensus in a synchronous message passing model with $n$ processes and up to $f$ failures, provided $n > 4f$. In the phase king algorithm, there are $f + 1$ phases, with 2 rounds per phase. Each process keeps track of its preferred output (initially equal to the process's own input value). In the first round of each phase each process broadcasts its own preferred value to all other processes. It then receives the values from all processes and determines which value is the majority value and its count. In the second round of the phase, the process whose id matches the current phase number is designated the king of the phase. The king broadcasts the majority value it observed in the first round and serves as a tie breaker. Each process then updates its preferred value as follows. If the count of the majority value the process observed in the first round is greater than $n/2 + f$, the process changes its preference to that majority value; otherwise it uses the phase king's value. At the end of $f + 1$ phases the processes output their preferred values.

Google has implemented a distributed lock service library called Chubby.[15] Chubby maintains lock information in small files which are stored in a replicated database to achieve high availability in the face of failures. The database is implemented on top of a fault-tolerant log layer which is based on the Paxos consensus algorithm. In this scheme, Chubby clients communicate with the Paxos *master* in order to access/update the replicated log; i.e., read/write to the files.[16]

Many peer-to-peer online real-time strategy games use a modified lockstep protocol as a consensus protocol in order to manage game state between players in a game. Each game action results in a game state delta broadcast to all other players in the game along with a hash of the total game state. Each player validates the change by applying the delta to their own game state and comparing the game state hashes. If the hashes do not agree then a vote is cast, and those players whose game state is in the minority are disconnected and removed from the game (known as a desync.)

Another well-known approach is called MSR-type algorithms which have been used widely in fields from computer science to control theory.[17][18][19]

| Source | Synchrony | Authentication | Threshold | Rounds | Notes |
|---|---|---|---|---|---|
| Pease-Shostak-Lamport [10] | Synchronous | Oral | $n > 3f$ | $f + 1$ | total communication $O(n^f)$ |
| Pease-Shostak-Lamport [10] | Synchronous | Written | $n > f + 1$ | $f + 1$ | total communication $O(n^f)$ |
| Ben-Or [20] | Asynchronous | Oral | $n > 5f$ | $O(2^n)$ (expected) | expected $O(1)$ rounds when $f < \sqrt{n}$ |
| Dolev et al.[21] | Synchronous | Oral | $n > 3f$ | $2f + 3$ | total communication $O(f^3 \log f)$ |
| Dolev-Strong [2] | Synchronous | Written | $n > f + 1$ | $f + 1$ | total communication $O(n^2)$ |
| Dolev-Strong [2] | Synchronous | Written | $n > f + 1$ | $f + 2$ | total communication $O(nf)$ |
| Feldman-Micali [22] | Synchronous | Oral | $n > 3f$ | $O(1)$ (expected) | |
| Katz-Koo [23] | Synchronous | Written | $n > 2f$ | $O(1)$ (expected) | Requires Public Key Infrastructure (PKI) |
| PBFT [24] | Asynchronous (safety) Synchronous (liveness) | Oral | $n > 3f$ | | |
| HoneyBadger [25] | Asynchronous | Oral | $n > 3f$ | $O(\log n)$ (expected) | per tx communication $O(n)$ - requires public-key encryption |
| Abraham et al.[26] | Synchronous | Written | $n > 2f$ | $8$ | |
| Byzantine Agreement Made Trivial [27][28] | Synchronous | Signatures | $n > 3f$ | $9$ (expected) | Requires digital signatures |

## Permissionless consensus protocols

Bitcoin uses proof of work, a difficulty adjustment function and a reorganization function to achieve permissionless consensus in its open peer-to-peer network. To extend bitcoin's blockchain or distributed ledger, *miners* attempt to solve a cryptographic puzzle, where probability of finding a solution is proportional to the computational effort expended in hashes per second. The node that first solves such a puzzle has their proposed version of the next block of transactions added to the ledger and eventually accepted by all other nodes. As any node in the network can attempt to solve the proof-of-work problem, a Sybil attack is infeasible in principle unless the attacker has over 50% of the computational resources of the network.

Other cryptocurrencies (e.g. Ethereum, NEO, STRATIS, ...) use proof of stake, in which nodes compete to append blocks and earn associated rewards in proportion to *stake*, or existing cryptocurrency allocated and locked or *staked* for some time period. One advantage of a 'proof of stake' over a 'proof of work' system, is the high energy consumption demanded by the latter. As an example, bitcoin mining (2018) is estimated to consume non-renewable energy sources at an amount similar to the entire nations of Czech Republic or Jordan, while the total energy consumption of Ethereum, the largest proof of stake network, is just under that of 205 average US households.[29][30][31]

Some cryptocurrencies, such as Ripple, use a system of validating nodes to validate the ledger. This system used by Ripple, called Ripple Protocol Consensus Algorithm (RPCA), works in rounds:

Step 1: every server compiles a list of valid candidate transactions;
Step 2: each server amalgamates all candidates coming from its Unique Nodes List (UNL) and votes on their veracity;
Step 3: transactions passing the minimum threshold are passed to the next round;

Step 4: the final round requires 80% agreement.[32]

Other participation rules used in permissionless consensus protocols to impose barriers to entry and resist sybil attacks include proof of authority, proof of space, proof of burn, or proof of elapsed time.

Contrasting with the above permissionless participation rules, all of which reward participants in proportion to amount of investment in some action or resource, proof of personhood protocols aim to give each real human participant exactly one unit of voting power in permissionless consensus, regardless of economic investment.[33][34] Proposed approaches to achieving one-per-person distribution of consensus power for proof of personhood include physical pseudonym parties,[35] social networks,[36] pseudonymized government-issued identities,[37] and biometrics.[38]

## Consensus number

To solve the consensus problem in a shared-memory system, concurrent objects must be introduced. A concurrent object, or shared object, is a data structure which helps concurrent processes communicate to reach an agreement. Traditional implementations using critical sections face the risk of crashing if some process dies inside the critical section or sleeps for an intolerably long time. Researchers defined wait-freedom as the guarantee that the algorithm completes in a finite number of steps.

The **consensus number** of a concurrent object is defined to be the maximum number of processes in the system which can reach consensus by the given object in a wait-free implementation.[39] Objects with a consensus number of $n$ can implement any object with a consensus number of $n$ or lower, but cannot implement any objects with a higher consensus number. The consensus numbers form what is called Herlihy's hierarchy of synchronization objects.[40]

| Consensus number | Objects |
|---|---|
| $1$ | atomic read/write registers, mutex |
| $2$ | test-and-set, swap, fetch-and-add, wait-free queue or stack |
| ... | ... |
| $2n-2$ | n-register assignment |
| ... | ... |
| $\infty$ | compare-and-swap, load-link/store-conditional,[41] memory-to-memory move and swap, queue with peek operation, fetch&cons, sticky byte |

According to the hierarchy, read/write registers cannot solve consensus even in a 2-process system. Data structures like stacks and queues can only solve consensus between two processes. However, some concurrent objects are universal (notated in the table with $\infty$), which means they can solve consensus among any number of processes and they can simulate any other objects through an operation sequence.[39]

## See also

- Uniform consensus
- Quantum Byzantine agreement
- Byzantine fault