Consensus (computer science)

From Wikipedia, the free encyclopedia

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 processes to agree on some data value that is needed during computation. Examples of applications of **consensus** include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts. The real world applications include clock synchronization, PageRank, opinion formation, power smart grids, state estimation, control of UAVs, load balancing and so on.

Contents

Problem description

The consensus problem requires agreement among a number of processes (or agents) for 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 somehow put forth their candidate values, communicate with one another, and agree on a single consensus value.

The consensus problem is a fundamental problem in control of 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 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 reached incorrectly.

Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes. These protocols must satisfy a number of requirements to be useful. For instance a trivial protocol could have all processes output binary value 1. This is not useful and thus the requirement is modified such that the output must somehow 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 and output a value only once and this decision is irrevocable. A process is called correct in an execution if it does not experience a failure. A consensus protocol tolerating halting failures must satisfy the following properties

Termination
: Every correct process decides some value.

Validity
: If all processes propose the same value v, then all correct processes decide v.

Integrity
: Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process.

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

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

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 failures 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 also 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 below

Termination
: Every correct process decides some value.

Validity
: If all correct processes propose the same value v, then all correct processes decide v.

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

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

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. Asynchronous versus synchronous models for message passing may be considered. 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.

A special case of the consensus problem called binary consensus restricts the input and hence the output domain to a single binary digit {0,1}. When the input domain is large relative to the number of processes, for instance an input set of all the natural numbers, it can be shown that consensus is impossible in a synchronous message passing model.

While real world communications are often inherently asynchronous it is more practical and useful to model synchronous systems.[1] In a fully asynchronous message-passing distributed system in which one process may have a halting failure, it has been proved that consensus is impossible.[2] However, this impossibility result derives from a worst-case scenario of a process schedule which is highly unlikely. In reality, process scheduling has a degree of randomness.[1]

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.

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.

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*: If all correct processes receive the same input value, then they must all output that value.
- *Strong validity*: For each correct process, its output must be the input of some correct process.
- *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:[3]

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.[4] 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.[5]

Solvability results for some agreement problems

There is a t-resilient anonymous synchronous protocol which solves the Byzantine Generals problem,[6][7] if t/n < 1/3 and the Weak Byzantine Generals case[4] where t is the number of failures and n is the number of processes.

For a system of 3 processors with one of them Byzantine, there is no solution for the consensus problem in a synchronous message passing model with binary inputs.[8]

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.[2] This result is sometimes called the FLP impossibility proof. The authors Michael J. Fischer, Nancy Lynch, and Mike Paterson were awarded a Dijkstra Prize for this significant work. The FLP result 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

An example of a polynomial time binary consensus protocol that tolerates Byzantine failures is the Phase King algorithm [9] by Garay and Berman. 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.[10] 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.[11]

Bitcoin uses proof of work to maintain consensus in its peer-to-peer network. Nodes in the bitcoin network attempt to solve a cryptographic proof-of-work problem, where probability of finding the solution is proportional to the computational effort, in hashes per second, expended, and the node that solves the problem has their version of the block of transactions added to the peer-to-peer distributed timestamp server accepted by all of the other nodes. As any node in the network can attempt to solve the proof-of-work problem, a Sybil attack becomes unfeasible unless the attacker has over 50% of the computational resources of the network.

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 from computer science to control theory.[12][13]

Applications of consensus protocols

One important application of consensus protocols is to provide synchronization. Traditional methods of concurrent access to shared data objects implement some form of mutual exclusion through locks. However the drawback is if a process dies while in its critical section, other correct processes may never acquire the lock. Thus mutual exclusion is poorly suited to asynchronous fault tolerant systems. A wait-free implementation of a data object supporting concurrent accesses guarantees that any process can complete its execution within a finite number of steps independent of the behavior of other processes. Atomic objects such as read/write registers have been proposed for the implementation of wait free synchronization. However it has been shown that such objects as well as traditional primitives such as test&set and fetch&add cannot be used for such an implementation.[14]

See also

- Uniform Consensus
- Quantum Byzantine agreement
- Byzantine fault tolerance

References

1. Aguilera, M. K. (2010). "Stumbling over Consensus Research: Misunderstandings and Issues". Lecture Notes in Computer Science **5959**: 59–72. doi:10.1007/978-3-642-11294-2_4. ISBN 978-3-642-11293-5.
2. Fischer, M. J.; Lynch, N. A.; Paterson, M. S. (1985). "Impossibility of distributed consensus with one faulty process" (PDF). *Journal of the ACM* **32** (2): 374–382. doi:10.1145/3149.214121.
3. Milosevic, Zarko; Martin Hutle; Andre Schiper (2009). "Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency". *Principles of Distributed Systems, Lecture Notes in Computer Science* **5293**: 300–314. doi:10.1007/978-3-642-10877-8_24.
4. Lamport, L. (1983). "The Weak Byzantine Generals Problem". *Journal of the ACM* **30** (3): 668. doi:10.1145/2402.322398.
5. Fischer, Michael J. "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)" (PDF). Retrieved 21 April 2014.
6. Lamport, L.; Shostak, R.; Pease, M. (1982). "The Byzantine Generals Problem" (PDF). *ACM Transactions on Programming Languages and Systems* **4** (3): 382–401. doi:10.1145/357172.357176.
7. Lamport, Leslie; Marshall Pease; Robert Shostak (April 1980). "Reaching Agreement in the Presence of Faults" (PDF). *Journal of the ACM* **27** (2): 228–234. doi:10.1145/322186.322188. Retrieved 2007-07-25.

8. Attiya, Hagit (2004). *Distributed Computing 2nd Ed*. Wiley. pp. 101–103. ISBN 978-0-471-45324-6.
9. Berman, Piotr; Juan A. Garay. "Cloture Votes: n/4-resilient Distributed Consensus in t + 1 rounds". *Theory of Computing Systems*. 2 **26**: 3–19. doi:10.1007/BF01187072. Retrieved 19 December 2011.
10. Burrows, M. (2006). *The Chubby lock service for loosely-coupled distributed systems* (PDF). Proceedings of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association Berkeley, CA, USA. pp. 335–350.
11. C., Tushar; Griesemer, R; Redstone J. (2007). *Paxos Made Live - An Engineering Perspective* (PDF). Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing. Portland, Oregon, USA: ACM Press New York, NY, USA. pp. 398–407. doi:10.1145/1281100.1281103. Retrieved 2008-02-06.
12. LeBlanc, Heath J. (April 2013). "Resilient Asymptotic Consensus in Robust Networks". *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*. doi:10.1109/JSAC.2013.130413.
13. Dibaji, S. M. (May 2015). "Consensus of second-order multi-agent systems in the presence of locally bounded faults". *Systems & Control Letters*. doi:10.1016/j.sysconle.2015.02.005.
14. Herlihy, Maurice. "Wait-Free Synchronization" (PDF). Retrieved 19 December 2011.

Further reading

- Herlihy, M.; Shavit, N. (1999). "The topological structure of asynchronous computability". *Journal of the ACM* **46** (6): 858. doi:10.1145/331524.331529.
- Saks, M.; Zaharoglou, F. (2000). "Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge". *SIAM J. Comput*. **29** (5): 1449–1483. doi:10.1137/S0097539796307698.

Categories: Distributed computing problems | Fault-tolerant computer systems

---