A cursory review of practical consensus protocols in distributed systems

Wentao Lu (002276355)

1 Abstract

In a distributed system, a group of replica server nodes is connected by a possibly unreliable network. These nodes could represent a number of processes without shared memory space, or even remote machines located far from each other. At the heart of such a system is the consensus problem, which deals with data consistency across all nodes. Consistency can be easily achieved in a distributed system that never malfunctions. However, this is impractical in the real world where many things can happen unexpectedly. For example, some nodes could crash or go awry, and some nodes are running too slow, network messages could be lost, delayed, duplicated or delivered out of order, etc. As a result, we need some consensus protocols that can correctly lead all nodes to reach an agreement even in the presence of these failures.

Among the many types of node failures, there are two extremes. In the simplest case of a failure-stop crash, a node always functions normally until it stops and will never restart again. On the other end of the spectrum, Lamport introduced the more complicated Byzantine failure where a node can even send malicious messages or misbehaves in an arbitrary manner while still operating. Conceptually, a Byzantine failure is the worst-case failure we may encounter, therefore, a BFT (Byzantine failure tolerance) consensus protocol is also a CFT (Crash failure tolerance) one, but the opposite does not hold. In this report, we attempt to survey some commonly used distributed consensus protocols mostly in the CFT context, while BFT often revolves around complicated smarter algorithms and requires much more time to study.

2 2PC - 2 Phase Commit Protocol

2PC [1] is the simplest form of atomic commitment protocol. In this protocol, a transaction is handled in two phases such that either all server nodes agree to commit or abort together. Distributed consensus is reached through the cooperation of one coordinator and multiple participants. The coordinator is responsible for coordinating all participants and decide to commit or abort the transaction, and a participant represents a node in the replica servers cluster. Each participant will notify the coordinator whether an operation succeeds or fails, then the coordinator will make the final judgment based on feedback from all participants and broadcast its decision.

In the voting phase, there are three steps:

- The coordinator first sends a prepare request to all participants to ask if they are ready to execute the transaction, and then waits for them to respond.

[1] lecture notes `https://courses.cs.washington.edu/courses/csep552/13sp/lectures/4/2PC.pdf`

- Each participant executes the transaction locally. On success, it writes undo and redo information to its log file, but does not commit. On failure, it immediately returns and quits the execution.
- All participants respond a *Yes* to the coordinator if execution has succeeded, which indicates that the transaction is committable, or a *No* if it has failed, which indicates that the transaction is not committable.

In the commit phase, there are two cases based on the outcome of the voting phase, if all participants responded *Yes* in the voting phase, then commit in four steps:

- The coordinator sends a commit request to all participants.
- On receiving the commit request, a participant commits the transaction and frees all related resources when it is done.
- After the transaction has been committed, participants send ACK to the coordinator.
- After receiving ACK from all participants, the coordinator completes the transaction.

Or else, if some participant responded *No* in the voting phase, then abort in four steps:

- The coordinator sends an abort request to all participants.
- On receiving the abort request, a participant aborts the transaction and frees all related resources when it is done.
- After the transaction has been aborted, participants send ACK to the coordinator.
- After receiving ACK from all participants, the coordinator completes the transaction.

The 2 Phase Commit Protocol can handle partial failures, but it is not very robust.[2] If a participant fails before receiving a prepare request, the coordinator will timeout and abort the transaction, in this case, all nodes remain in consensus, but availability is undermined since the system cannot commit whatsoever. If a participant fails right after receiving a commit request, the coordinator will timeout without receiving ACK from it, in this case, data is consistent except when that participant recovers without updates. However, if the coordinator fails in either phase, the system becomes unavailable, client requests cannot be served anymore, any participant waiting for it to respond will block, the outcome is undefined until the coordinator recovers. In the worst case, the coordinator crashes after sending only part of the commit requests, so that some participants will commit while some others will not, which can lead to severe data inconsistency.

From the behavior of 2PC, it's easy to see that consensus can be reached as long as the coordinator does not crash. Since a commit decision can only be made when all participants voted *Yes*, that same decision will be committed on every node, the protocol is safe. 2PC is easy to understand and implement, but it cannot solve the single node failure which may cause corrupt data. The coordinator matters so much that once it's down the whole system gets stuck. There are two rounds of message exchange between the coordinator and participants, results in a total of $4n - 4$ messages, so the complexity of this protocol is $O(n)$ in the general case.

[2]lecture notes `https://www2.cs.duke.edu/courses/fall07/cps212/consensus.pdf`

3 3PC - 3 Phase Commit Protocol

3PC is an extension of 2PC that aims to solve the blocking problem when the coordinator crashes. As a simple fix, another intermediate *pre-commit* phase is added between the two phases of 2PC to ensure that all participants are in consensus before the final stage.[1]

In the *canCommit* phase, there are two steps:[3]

- The coordinator first sends a *canCommit* request to all participants to ask if they are able to commit the transaction and then waits for them to respond.
- On receiving the *canCommit* request, a participant responds *Yes* if the transaction is deemed as executable, or *No* otherwise.

In the *preCommit* phase, there are two cases: pre-commit or abort.

- If all participants responded *Yes* in the previous phase, the coordinator sends a *preCommit* request to them and enters the *prepare* state. On receiving such a request, a participant executes the transaction locally. On success, it writes undo and redo information to its log file, responds ACK to the coordinator. Otherwise, it immediately returns and quits the execution.
- If some participant responded *No* in the previous phase, the coordinator sends an *Abort* request to all participants. Then, a participant aborts the transaction either when receiving it or timeouts.

In the *doCommit* phase, there are two cases: commit or abort.

- If all participants responded ACK in the previous phase, the coordinator switches from the *prepare* state to the *commit* state and sends a *doCommit* request to all of them. On receiving such a request, a participant commits the transaction, frees all related resources when it is done, and responds to the coordinator with ACK . Once the coordinator receives ACK from all participants, it completes the transaction.
- If the coordinator did not receive ACK from some participant in the previous phase (perhaps it received something else or timeout has reached), it sends an *Abort* request to all participants. On receiving it, a participant aborts the transaction and responds to the coordinator with another ACK . Once the coordinator receives ACK from all participants, it completes the transaction.

Once the protocol enters the *doCommit* phase, if the coordinator fails or the network between it and the participants fails, then the participants will not be able to receive a *doCommit* or *Abort* request. In this case, a participant will move on to commit the transaction by itself after timeouts. In contrast with 2PC , 3PC solves the blocking problem when the coordinator is down. This is because once a participant failed to receive a *doCommit* from the coordinator, it always commits by default without blocking and waiting. The correctness and complexity analysis of 2PC can be applied to 3PC in a similar fashion. However, 3PC still suffers from the problem of inconsistent data in some cases. For instance, if a participant fails to receive an *Abort* request due to unreliable network and commits itself after the timeout, its local data will deviate from other participants who have received that request.

[3]lecture notes `https://www.cs.cornell.edu/courses/cs5412/2012sp/slides/XI-2PCand3PC.pdf`

4 Variants of 2PC & 3PC

While 3PC solved the blocking issue when the coordinator is down, it also introduced another phase that comes with extra communication overhead. Besides, there are many other alternative variants of 2PC and 3PC that addressed the same problem but tries to maintain the message complexity.

One such example is the RL2PC protocol published in 2012[2], which is just another improved version of 2PC. It introduces a new node to play the role of a sub-coordinator. When 2PC is working normally, the sub-coordinator is responsible for communicating with the participants and the completion of a transaction. It acts just as a coordinator but also works to send copies of its log information to the primary coordinator. The primary coordinator keeps itself synchronized with the sub-coordinator based on the messages and logs it has received, but it does not talk to the participants directly. In case the coordinator or sub-coordinator crashes, the other one can simply take over its place and continue to coordinate the participants since it is updated and has the full control privilege. Once the failed coordinator recovers, it will then provide a copy of the most up-to-date data log to make sure that the failed one can get updates and resume normal work. From the perspective of participants, at any given time, it seems that there's only one coordinator.

The difference between RL2PC is that additional space is required to allow for another coordinator node. Obviously, this approach can be further generalized to more coordinators, where we can sacrifice storage in exchange for system availability. The complexity of this algorithm is the same as 2PC since we don't need to send another round of messages. However, the system is much more robust, it will only become unavailable if both coordinators die at the same time, but the chances are rare.

Another non-blocking version of 2PC is the EasyCommit protocol[3]. Instead of adding nodes to the cluster to ensure system safety, EasyCommit accomplishes the same goal by improving the commit phase of 2PC. In this updated version of the commit phase, a participant who has received a global decision from the coordinator will first broadcast his received message to all the other nodes before it commits or aborts. If a participant has received a global decision from other participants, then it does not have to wait for the coordinator. Consequently, unless all nodes have crashed, a participant can always receive a decision message from its peers so that a failed coordinator will not block the system. Despite the fact that we have added possibly redundant messages into the system, unlike 3PC, participants now do not need another round of message waiting. Once it received the message from any other node, it can safely ignore other messages and proceed, so that the overhead is trivial.

Moreover, RL2PC also solves the single point failure of the coordinator. If the coordinator did not receive messages from a participant before timeouts, it aborts the transaction. On the other hand, if a participant did not receive messages from the coordinator or any other peer before timeouts, it will attempt to communicate with other peers to reach an agreement. (This happens most likely when the coordinator crashed early so that no participant can receive messages from it and broadcast) For example, if the coordinator died right after sending a request to participant A, other participants can still get to know about the global decision from A, then all of them will be in consensus and take the same action. If A also died before broadcasting its received message, which means that A did not commit, then all the other participants will eventually timeout and lose connection with A, in this case they can randomly select a new active coordinator only among themselves and abort. In all situations, consensus is well maintained and the system is Crash failure tolerant.

5 Raft

Among all the famous consensus protocols, Raft[4] is the easiest to understand. Raft sees every server as a state machine, and each state machine maintains a log file. Whenever a client submits a request to the server, it is equivalent to appending a log entry to the state machine. Raft algorithm ensures that all the log files are eventually synchronized in a distributed cluster so that all state machines will handle requests by taking actions in the same order, and output the same final result.

In this protocol, a server process can be classified into three roles: leader, candidate, and follower. At any given time, a server can only act as one of them, but this role evolves over time. Later at another time, that server can play a different role. The leader election phase is very much analogous to the real world election as to how several followers vote a leader: at first there are no leaders, everyone in the cluster is a follower, when a round of vote starts, everyone is eligible to become the candidate, and then a leader will be elected, his domination starts with a fixed term or lease, before the term expires, everyone else becomes his followers to accept his dominance.

The key step of Raft to reach consensus is leader election. When the system starts, all nodes are initially followers blocking within a timeout, every follower has a random election timeout within a reasonable range. The first follower who wakes up from the timeout will initiate a round of the vote, he votes himself, increments his term and keeps track of it, then he requests other nodes which are still sleeping in the timeout to vote for him. As other nodes wake up and vote, once this candidate has received more than half votes, he wins the vote and will change his role to a leader. Then he starts to broadcast his heartbeat periodically to announce the beginning of his term, so that no other node will initiate another round of the vote. If a candidate in the voting phase receives a heartbeat from other nodes whose term is higher than his, it means that some other candidate has already won the vote so he just gives up. It is possible that many candidates could wake up and start rounds of vote at the same time, but none of them is likely to receive the majority of votes, in this case, candidates will finally timeout while waiting for the votes, and whoever timeouts first will initiate a new round of the election. Since the election timeout is at random, Raft guarantees that the election phase can reach an agreement very quickly. A leader will continue to play his role until crash failure, if this happened, followers would not receive heartbeats since the leader has died, then, the follower who timeouts first and realizes this fact will initiate a new vote phase for himself.

As it is true with the coordinator in 2PC and 3PC , the leader role in Raft is crucial as well. Raft's leader election procedure warranties the availability of a leader even in the presence of a leader crash, which renders us more robustness. In fact, safety is also naturally inherited in this algorithm. Within any term, there can be exactly one leader, and only the server which has the longest term (server which is synchronized with the most recent log entries) can be elected as the leader. Therefore, the leader is always up-to-date, so any message passing mechanism that follows can easily realize data consistency.

Now that we have an up-to-date leader, Raft simply adopts 2PC for the interaction between server nodes. When a client issues a request to the cluster leader, the state of data at the leader node is *uncommitted*. The leader then concurrently forwards data to all the followers and wait for their responses. After the majority of followers have acknowledged the receipt of data, the leader then responds ACK to the client. Once the acknowledgment has been replied to the client, the state of data transfers from *uncommitted* to *committed*, and the leader will send a commit request to all followers, who then commit data locally and respond ACK to the leader. It is plain to see that this is indeed 2PC , what differs is that Raft's leader is always available so that blocking is not possible thanks to the efficient election mechanism. If some followers have crashed, consensus is still observed among all the active nodes. Therefore, problems can only emerge during the vulnerable period when the old leader has died but a new leader has not yet been voted.

If leader crashes before receiving the client's request, no ACK will be responded so the client has to retry after timeouts, data consensus is not affected, and a new leader will take over when the client retries. If leader crashes before forwarding data to the follower, the state of data is *uncommitted* so ACK is not expected, again the client has to retry after timeouts. In both cases, data is consistent, when the crashed leader recovers, it becomes a follower of the new leader and synchronizes data from it.

If data has already been forwarded to all the followers, but leader crashes before followers respond with ACK , then the state of data at the follower nodes are *uncommitted*, but they are consistent, data can be committed after the system selects a new leader. In this case, the client has no idea if data has been committed so it retries after a few seconds, which may cause duplicated commits. Fortunately, Raft also enforces a remote process call to be idempotent so that duplicated client requests do not affect data consistency. Likewise, if leader crashes after forwarding data to only some followers, then the state of data at the follower nodes are *uncommitted* and inconsistent. In this case, a leader can only be selected among the followers who have already received that data since Raft requires a qualified leader to be up-to-date, after vote, the new leader will synchronize data to the other followers, so that data won't be lost and eventually it's consistent.

In summation, eventual consensus can be achieved in all possible cases of fail-stop failure. The Raft election mechanism is blazingly fast, and inconsistent data can take place only within a very short window. Speaking of message complexity, Raft runs in $O(n)$ time in general since it adopts the 2PC as mentioned above.

6 Paxos

Another leader-based consensus protocol is the Paxos algorithm which was proposed by Lamport in 1980[5]. Paxos is widely cited and regarded as the most efficient distributed consensus algorithm we have so far. Nevertheless, it is often thought to be abstruse and difficult to implement despite the attractive story the author creates to illustrate his ideas.[4] As a result, Lamport translated his original paper into a much simpler and readable version with descriptive steps of proof in 2001.[6]

As mentioned in his refined paper, there are three roles in this algorithm: proposer, acceptor, and learner. A proposer can propose a proposal, an acceptor can accept or reject a proposal, once a proposal has been accepted, its value is chosen and cannot be mutated. Acceptors broadcast a chosen proposal to the learner, then the learner learns that chosen value. Similar to the Raft algorithm, a server can play different roles depending on the context. What makes Paxos different is that it allows multiple proposers to concurrently make proposals, while in Raft, there can only be one leader at any given time.

Paxos has two phases, the *prepare* phase and the *acceptor* phase. In the *prepare* phase, a proposer first selects a non-decreasing number as the proposal ID, then it sends a prepare request with ID n to some majority group of acceptors. In receipt of the prepare request, an acceptor checks its history, if it has never responded to any prepare request whose ID is greater than or equal to n, then it responds to the proposer with the max-ID proposal it has ever received (or *null*), and promises that it won't respond to any prepare request whose ID is less than n. If the acceptor has already promised another proposal with a larger ID, it does not respond.

[4]slides `https://courses.cs.washington.edu/courses/csep552/13sp/lectures/5/paxos.pdf`

In the *acceptor* phase, the proposer waits for responses from acceptors. If more than half acceptors have responded, it then sends an accept request to these acceptors, which includes the ID n and a value. That value is the value of the largest-ID proposal that it receives from the acceptors, or an arbitrary value if it receives no proposals. In receipt of the accept request, an acceptor accepts the proposal if it has not responded to any proposal with a larger ID, or rejects the proposal if it does.

To derive the correctness of this abstruse algorithm, we need to step back and start from the goals of consistency: if no proposals have been proposed, then no proposal will be chosen, if some proposals have been proposed, only one of them will eventually be chosen. Simply put, there must be one and only one proposal to be chosen. In order to enforce that at most one proposal can be chosen, that proposal needs to be accepted by the majority of acceptors. This is because any two majority groups or quorums must share at least one node, so that two distinct majority groups cannot accept two different proposals, otherwise they cannot both be majority. Besides, if we want to ensure that at least one proposal will be chosen, an acceptor must accept the first proposal it receives, otherwise majority cannot be guaranteed. This leads us to the first condition:

P1 an acceptor must accept the first proposal it receives.

However, it is still possible that majority is not met when there are many different proposals, we need to find a way to distinguish between proposals with the same value. To do so, a proposal can be prefixed with a distinct ID number, so that we can choose different proposals. But we still require the chosen proposal to be unique, now we need another condition:

P2 if a proposal with value v is chosen, all chosen proposals with a higher ID number must also have value v.

Since a chosen proposal must be accepted by some acceptors, this can be rephrased as:

P2a if a proposal with value v is chosen, all proposals with a higher ID number that is accepted by any acceptor must also have value v.

This time we have another corner case where some acceptor c has not yet received any proposal, but a new proposal with a higher ID and a value $\neq v$ could be sent to c. In this case, c must accept this proposal according to P1, which then violates P2a, so we need to further modify P2a as:

P2b if a proposal with value v is chosen, any proposal with higher ID number proposed by any proposer must also have value v.

The causal relationship of "propose $\Rightarrow$ accept $\Rightarrow$ choose" implies that "P2b $\Rightarrow$ P2a $\Rightarrow$ P2", so in order to realize P2b, we just need to meet a variant of it:

P2c if a proposal with value v and ID n is proposed, there must be a majority set of acceptors S such that one of the following is satisfied:

1 Either: S does not have any acceptor who has accepted a proposal with ID less than n.

2 Or: the proposal accepted by acceptors in S with the highest ID has value v.

To maintain P2c, when a proposer proposes a proposal with ID n, it must also learn the value of the proposal with the highest ID $\leq n$. Such a proposal may have already been accepted or not yet, in the latter case the proposer is not able to predict and learn an unaccepted value. Therefore, the proposer must also request the acceptor not to accept any proposals whose ID is less than n, this leads to an even stricter version of P1:

P1a an acceptor can accept a proposal with ID n, if and only if it has not responded to any proposal whose ID is greater than n.

Putting all the pieces together, we can now verify that these conditions are actually in alignment with what the Paxos two phases do. Thus, Paxos ensures safety and correctness.

7 Conclusion

2PC is the simplest version of consensus protocols. It is concise, straightforward, widely used and easy to implement. However, it also suffers from the blocking issue and single point of failure. As an extension of 2PC to address these problems, 3PC introduces an extra phase of message exchange with additional communication overhead, as a result blocking is alleviated, but in some cases data inconsistency could still occur. Meanwhile, many invariants of 2PC and 3PC have been researched to deal with the lack of robustness of the coordinator node, but these models always come at a price, either redundant messages are necessary between participants, or additional storage is required to act as backup coordinators. These approaches alone are conservative inside that cannot effectively solve the consensus problem in real-world large applications.

Paxos is the most classic consensus algorithm which is widely recognized in theory. By allowing any node to serve as a leader or proposer to avoid a single vulnerable coordinator, it is non-blocking, crash failure tolerant, network partition tolerant and is able to efficiently solve the consistency problem in modern distributed systems. The only downside is its opacity that makes it challenging to implement, and the mathematical proof is somewhat abstract. Hence, in pursuit of an easily understandable consensus algorithm, another famous algorithm Raft was invented in 2014 that comes to the rescue, which is as efficient as Paxos. Compared to Paxos, Raft emphasizes the role of a single leader, it divides the key elements of the consensus procedure into different components such as leader election and log replication, this makes it much easier for students to learn. In addition, the authors of Raft also draft the various components in much detail so that implementation is comfortable.

8 Future Improvements

None of the protocols we have surveyed this time accounts for Byzantine failures, they are safe and robust to some extent but only in the presence of fail-stop faults. While these CFT protocols are the building blocks of distributed consensus algorithms, there are also many BFT protocols such as the recursive oral message algorithm $OM(m)$[7], as well as many other improved variants of CFT protocols including Byzantizing Paxos, multi-Paxos, BFT Raft Tangaroa, Zab, viewstamped replication and so on. For future study, we plan to conduct a more detailed review of consensus protocols and failure detectors that are capable of dealing with exotic node faults.

References

[1] I. Keidar and D. Dolev, "Increasing the resilience of distributed and replicated database systems," *Journal of Computer and System Sciences*, vol. 57, pp. 309–324, Dec. 1998. DOI: 10.1006/jcss.1998.1566.

[2] X. Yan, J. Yang, and Q. Fan, “An improved two-phase commit protocol adapted to the distributed real-time transactions,” *Przeglad Elektrotechniczny*, vol. 88, pp. 27–30, Jan. 2012.

[3] S. Gupta and M. Sadoghi, “Easycommit: A non-blocking two-phase commit protocol,” in *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, Eds., OpenProceedings.org, 2018, pp. 157–168. DOI: `10.5441/002/edbt.2018.15`. [Online]. Available: `https://doi.org/10.5441/002/edbt.2018.15`.

[4] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319, ISBN: 978-1-931971-10-2. [Online]. Available: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[5] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 133169, May 1998, ISSN: 0734-2071. DOI: `10.1145/279227.279229`. [Online]. Available: `https://doi.org/10.1145/279227.279229`.

[6] ——, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[7] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *Journal of the Association for Computing Machinery 27*, vol. 2, 1980, 2005 Edsger W. Dijkstra Prize in Distributed Computing. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/reaching-agreement-presence-faults/`.