Consensus and Coordination in Distributed Systems

Arjun Reddy Lingala

arjunreddy.lingala@gmail.com

Abstract

Distributed systems serve as the foundation of contemporary computing infrastructures, powering applications ranging from cloud platforms and blockchain networks to large- scale data management and real-time analytics. One of the fundamental challenges in distributed computing is ensuring that multiple autonomous nodes, often operating under unpredictable network conditions, maintain a coherent and agreed-upon system state. This paper provides a thorough exploration of consistency and consensus, two essential principles that uphold the correctness and reliability of distributed systems. Consistency, which can be formally defined through models such as linearizability, sequential consistency, and eventual consistency, determines how system state is perceived across different nodes over time. We examine the trade-offs between strong and weak consistency models, their impact on system efficiency, and the mechanisms by which distributed databases and replicated state machines uphold these guarantees, even in the presence of network failures, partitioning, and latency constraints. Consensus, the process that allows distributed nodes to agree on a single value or decision, is explored through foundational algorithms such as Paxos [3], Raft [4], and Byzantine Fault Tolerant protocols [5]. We provide an in-depth analysis of the critical properties of these algorithms, including safety, and fault tolerance, and discuss their practical implications in real-world systems. Additionally, we investigate the CAP theorem [7] and its significance in balancing consistency and availability, shedding light on how distributed architectures navigate these inherent trade-offs. This paper also covers recent advancements in consensus protocols, including blockchain-based mechanisms such as proof-of-work and proof-of-stake, along with hybrid approaches, evaluating their performance, security, and scalability characteristics. We conclude by identifying open research challenges and future directions, encompassing eventual consensus strategies, decentralized trust frameworks, and AI- driven techniques for consistency management.

Keywords: Distributed Computing, Consistency Models, Consensus Protocols, Paxos, Raft, Byzantine Fault Tolerance, CAP Theorem, Blockchain Consensus, Fault-Tolerant Systems, Data Replication.

I. INTRODUCTION

Most replicated databases provide at least eventual consistency, which means that if we stop writing to the database and wait for some unspecified length of time, then eventually all read requests will return the same value. In other words, the inconsistency is temporary, and it eventually resolves itself. However, this is a very weak guarantee which doesn't say anything about when the replicas will converge. Until the time of convergence, reads could return anything or nothing. For ex- ample, if we write a value and then immediately read it again, there is no guarantee that we will see the value we just wrote, because the read

may be routed to a different replica. Eventual consistency is hard for application developers because it is so different from the behavior of variables in a normal single- threaded program. If we assign a value to a variable and then read it shortly afterward, we don't expect to read back the old value, or for the read to fail. A database looks superficially like a variable that we can read and write, but in fact it has much more complicated semantics. When working with a database that provides only weak guarantees, we need to be constantly aware of its limitations and not accidentally assume too much. Bugs are often subtle and hard to find by testing, because the application may work well most of the time. The edge cases of eventual consistency only become apparent when there is a fault in the system or at high concurrency. Systems with stronger guarantees may have worse performance or be less fault-tolerant than systems with weaker guarantees. Nevertheless, stronger guarantees can be appealing because they are easier to use correctly. There is some similarity between distributed consistency models and the hierarchy of transaction isolation levels we discussed previously. But while there is some overlap, they are mostly independent concerns, transaction isolation is primarily about avoiding race conditions due to concurrently executing transactions, whereas distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

II. LINEARIZABILITY

In an eventually consistent database, if we ask two different replicas the same question at the same time, we may get two different answers. The exact definition of linearizability is quite subtle, but the basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic. With this guarantee, even though there may be multiple replicas in reality, the application does not need to worry about them. In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written. Maintaining the illusion of a single copy of the data means guaranteeing that the value read is the most recent, up-to-date value, and doesn't come from a stale cache or replica. Lineariz- ability is easily confused with serializability, as both words seem to mean something like can be arranged in a sequential order. However, they are two quite different guarantees, and it is important to distinguish between them. Serializability is an isolation property of transactions, where every transaction may read and write multiple objects. It guarantees that transactions behave the same as if they had executed in some serial order. It is okay for that serial order to be different from the order in which transactions were actually run. Linearizability is a recency guarantee on reads and writes of a register. It doesn't group operations together into transactions, so it does not prevent problems such as write skew, unless we take additional measures such as materializing conflicts. A database may provide both serializability and linearizability, and this combination is known as strict serializability or strong one- copy serializability. However, serializable snapshot isolation is not linearizable, by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

A system that uses single-leader replication needs to ensure that there is indeed only one leader, not several. One way of electing a leader is to use a lock, every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader. No matter how this lock is implemented, it must be linearizable, all nodes must agree which node owns the lock; otherwise it is useless. Coordination services like Apache ZooKeeper [1] and etcd [2] are often used to implement distributed locks and leader election. They use consensus algorithms to implement linearizable operations in a fault-tolerant way. There are still many subtle details to implementing locks and leader election correctly, and libraries like Apache Curator [5] help by providing higher-level recipes on top of ZooKeeper [1]. However, a

linearizable storage service is the basic foundation for these coordination tasks. Distributed locking is also used at a much more granular level in some distributed databases, such as Oracle Real Application Clusters. RAC uses a lock per disk page, with multiple nodes sharing access to the same disk storage system. Since these linearizable locks are on the critical path of transaction execution, RAC deployments usually have a dedicated cluster interconnect network for communication between database nodes. Since linearizability essentially means behave as though there is only a single copy of the data, and all operations on it are atomic, the simplest answer would be to really only use a single copy of the data. However, that approach would not be able to tolerate faults, if the node holding that one copy failed, the data would be lost, or at least inaccessible until the node was brought up again. The most common approach to making a system fault-tolerant is to use replication. In a system with single-leader replication, the leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes. If we make reads from the leader, or from synchronously updated followers, they have the potential to be linearizable. However, not every single- leader database is actually linearizable, either by design or due to concurrency bugs. Some consensus algorithms, bear a resemblance to single-leader replication. However, consensus protocols contain measures to prevent split brain and stale replicas. Systems with multi-leader replication are generally not linearizable, because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes. For systems with leaderless replication, people some- times claim that we can obtain strong consistency by requiring quorum reads and writes. Depending on the exact configuration of the quorums, and depending on how we define strong consistency, this is not quite true. Last write wins conflict resolution methods based on time-of-day clocks are almost certainly nonlinearizable, because clock timestamps cannot be guaranteed to be consistent with actual event ordering due to clock skew. Sloppy quorums also ruin any chance of linearizability.

As some replication methods can provide linearizability and others cannot, it is interesting to explore the pros and cons of linearizability in more depth. With a multi-leader database, each datacenter can continue operating normally since writes from one datacenter are asynchronously replicated to the other, the writes are simply queued up and exchanged when network connectivity is restored. On the other hand, if single-leader replication is used, then the leader must be in one of the datacenters. Any writes and any linearizable reads must be sent to the leader, for any clients connected to a follower datacenter, those read and write requests must be sent synchronously over the network to the leader datacenter. If the network between datacenters is interrupted in a single-leader setup, clients connected to follower datacenters cannot contact the leader, so they cannot make any writes to the database, nor any linearizable reads. They can still make reads from the follower, but they might be stale. If the application requires linearizable reads and writes, the network interruption causes the application to become unavailable in the datacenters that cannot contact the leader. If clients can connect directly to the leader datacenter, this is not a problem, since the application continues to work normally there. But clients that can only reach a follower datacenter will experience an outage until the network link is repaired.

III. ORDERING AND CAUSALITY

Causality imposes an ordering on events: cause comes before effect, a message is sent before that message is received, the question comes before the answer. One thing leads to another, one node reads some data and then writes something as a result, another node reads the thing that was written and writes something else in turn, and so on. These chains of causally dependent operations define the causal order in the system, what happened before what. If a system obeys the ordering imposed by causality, we say that it is causally consistent. For example, snapshot isolation provides causal consistency: when we read from the database, and we see some piece of data, then we must also be

able to see any data that causally precedes it. A total order allows any two elements to be compared, so if we have two elements, we can always say which one is greater and which one is smaller. The difference between a total order and a partial order is reflected in different database consistency models. *Linearizability* – In a linearizable system, we have a total order of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first. *Causality* – We said that two operations are concurrent if neither happened before the other. Two events are ordered if they are causally related, but they are incomparable if they are concurrent. This means that causality defines a partial order, not a total order, some operations are ordered with respect to each other, but some are incomparable. Any system that is linearizable will preserve causality correctly. In particular, if there are multiple communication channels in a system, linearizability ensures that causality is automatically preserved without the system having to do anything special. The fact that linearizability ensures causality is what makes linearizable systems sim- ple to understand and appealing, but making a system linearizable can harm its performance and availability, especially if the system has significant network delays. Some distributed data systems have abandoned linearizability, which allows them to achieve better performance but can make them difficult to work with. Linearizability is not the only way of preserving causality. A system can be causally consistent without incurring the performance hit of making it linearizable. Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures. In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently. Based on this observation, researchers are exploring new kinds of databases that preserve causality, with performance and availability characteristics that are similar to those of eventually consistent systems. In order to maintain causality, we need to know which operation happened before which other operation. This is a partial order, concurrent operations may be processed in any order, but if one operation happened before another, then they must be processed in that order on every replica. Thus, when a replica processes an operation, it must ensure that all causally preceding operations have already been processed and if some preceding operation is missing, the later operation must wait until the preceding operation has been processed.

IV. SEQUENCE NUMBER ORDERING

Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impractical. In many applications, clients read lots of data before writing something, and then it is not clear whether the write is causally dependent on all or only some of those prior reads. Explicitly tracking all the data that has been read would mean a large overhead. We can use sequence numbers or timestamps to order events. A timestamp need not come from a time-of-day clock. It can instead come from a logical clock, which is an algorithm to generate a sequence of numbers to identify operations, typically using counters that are incremented for every operation. Such sequence numbers or timestamps are compact, and they provide a total order which is every operation has a unique sequence number, and we can always compare two sequence numbers to determine which is greater. In particular, we can create sequence numbers in a total order that is consistent with causality. In a database with single-leader replication, the replication log defines a total order of write operations that is consistent with causality. The leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log. If a follower applies the writes in the order they appear in the replication log, the state of the follower is always causally consistent.

A. Sequence Number Generators

When there are more than a single leader, it is less clear how to generate sequence numbers for operations. Various types of methods are used which include: Each node can generate its own independent set of sequence numbers. If we have two nodes, one node can generate only odd numbers and the other only even numbers. In general, we could reserve some bits in the binary representation of the sequence number to contain a unique node identifier, and this would ensure that two different nodes can never generate the same sequence number. We can attach a timestamp from a time-of-day clock to each operation. Such timestamps are not sequential, but if they have sufficiently high resolution, they might be sufficient to totally order operations. We can pre-allocate blocks of sequence numbers. Each node claims a range of numbers and can independently assign sequence numbers from its block, and allocate a new block when its supply of sequence numbers begins to run low. These options perform better and are more scalable than pushing all operations through a single leader that increments a counter. They generate a unique, approximately increasing sequence number for each opera- tion. However, the sequence numbers they generate are not consistent with causality. Causality problems occur because these sequence number generators don't correctly capture the ordering of operations across different nodes. Each node may process a different number of operations per second. Thus, if one node generates even numbers and the other generates odd numbers, the counter for even numbers may lag behind the counter for odd numbers, or vice versa. If we have an odd-numbered operation and an even-numbered operation. Timestamps from physical clocks are subject to clock skew, which can make them inconsistent with causality. In the case of the block allocator, one operation may be given a sequence number in the range A in the range and a causally later operation may be given a number from the different range.

B. Lamport Timestamps

Lamport Timestamps [8] is very simple method for generat- ing sequence numbers that is consistent with causaulity. Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The Lamport timestamp [8] is then simply a pair of counter, node ID. Two nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique. A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering, if we have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp. Every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum. As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp. Lamport timestamps are sometimes confused with version vectors, Although there are some similarities, they have a different purpose: version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas Lamport timestamps always enforce a total ordering. From the total ordering of Lamport times- tamps, we cannot tell whether two operations are concurrent or whether they are causally dependent. The advantage of Lamport timestamps over version vectors is that they are more compact.

V. CONCLUSION

The investigation of coordination and consensus in distributed systems highlights their critical importance in constructing resilient, scalable, and fault-tolerant computational frameworks within today's interconnected digital infrastructure. As distributed architectures span diverse applications from

decentralized cloud platforms and blockchain ecosystems to edge computing grids and IoT deployments, the challenge of achieving unified, consistent, and timely agreement among geographically distributed and heterogeneous components re- mains central to modern system engineering. This study has rigorously examined the theoretical foundations, algorithmic advancements, and pragmatic considerations that define state- of-the-art consensus methodologies, tracing their progression from classical frameworks such as Paxos and Raft to modern adaptations like Byzantine Fault Tolerance and blockchain- derived approaches. A critical synthesis of the literature reveals that consensus protocols must navigate a complex interplay of competing requirements: ensuring safety and livness, while balancing latency, throughput, and resource efficiency. The CAP theorem [7] which posits the impossibility of simultaneously achieving Consistency, Availability, and Partition Tolerance—continues to serve as a guiding principle, compelling system designers to make context-dependent compromises. For instance, strongly consistent systems like ZooKeeper and etcd prioritize linearizability at the expense of availability during partitions, whereas eventually consistent systems like Dynamo and Cassandra favor availability and partition resilience, deferring consistency to asynchronous reconciliation. Meanwhile, the emergence of hybrid models, such as Google's Spanner and CockroachDB, demonstrates the feasibility of leveraging synchronized clocks and hybrid logical clocks to mitigate the CAP trade-offs in geo-replicated environments. As distributed systems continue to underpin critical digital infrastructure—from cloud computing and financial systems to IoT networks and decentralized applications, the principles of consistency and consensus remain paramount. While ex- isting algorithms and models provide robust foundations, the ever-growing scale, complexity, and heterogeneity of modern distributed environments necessitate continued innovation in this domain. Future research should focus on bridging the gap between theoretical impossibility results and practical engineering solutions, developing adaptive and intelligent dis- tributed coordination techniques, and exploring the impact of emerging paradigms such as edge computing, blockchain, and AI-driven optimizations.

REFERENCES

[1]C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model-Based API Testing of Apache ZooKeeper," in Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE- C), Buenos Aires, Argentina, 2017, pp. 373–375.

[2]"etcd 2.0.12 Documentation," CoreOS, Inc., 2015.

[3]Leslie Lamport: "Paxos Made Simple," ACM SIGACT News, volume 32, number 4, pages 51–58, December 2001.

[4]J. Liu, Y. Zhang, and X. Li, "Improving Raft When There Are Failures," in Proceedings of the 2019 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, Portugal, 2019, pp. 1–10.

[5]"Apache Curator," Apache Software Foundation, curator.apache.org, 2015.

[6]Y. Mao, F. P. Junqueira, and K. Marzullo, "Dynamic Practical Byzantine Fault Tolerance," in Proceedings of the 2008 International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), Anchorage, AK, USA, 2008, pp. 287–296.

[7]S. Gilbert and N. A. Lynch, "Perspectives on the CAP Theorem," Computer, vol. 45, no. 2, pp. 30–36, Feb. 2012. doi:10.1109/MC.
2011.389

[8]L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, vol. 21, no. 7, pp. 558–565,
July 1978. doi:10.1145/359545.359563

[9]J. Gray and L. Lamport, ”Consensus on Transaction Commit,” ACM Transactions on Database Systems, vol. 31, no. 1, pp. 133–160, Mar. 2006. doi:10.1145/1132863.1132867

[10] M. Herlihy, ”Wait-Free Synchronization,” ACM Transactions on Programming Languages and Systems, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[11] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, ”Challenges to Adopting Stronger Consistency at Scale,” in Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV), 2015.