**Definitions:**

* **Latency**: The amount of time needed between the sending of the request and receiving the response.
* **Bandwidth**: The amount of data transferred between two point per unit of time.
* **Availability**: The ability of the service to response correctly to requests within a certain time.
* **Fault Tolerance**: Making the system work even if some parts of the system are not working. This is caused by node crashes or network partitions. And we can achieve it by getting better reliable hardware and redundancy.
* **Network partition:**  some links are dropping or delaying the messages for extended period of time.

**System Models:**

* **Network Models**:
  + **Reliable:** the messages will arrive if and only if they have been sent. May need reordering.
  + **Fair Loss:** Messages can get lost or delayed but if we try enough the messages will arrive.
  + **Arbitrary:**  the links are not secure and there might be an active adversary that modifies the data in the link or just blocking the link.
* **Node Models:**
  + **Crashe-Fail:** If node is crashed it will stop working indefinitely
  + **Crashe-recovery:** If node fails it will stop working for a period of time and then rejoin the system
  + **Byzantine:** the node might deviate from the original algorithm and there might be some kind of adversary that manipulates it and returns wrong results.
* **Time Models:**
  + **Synchronous:** There is an upper limit for the message latency in the network.
  + **Semi-Synchronous:** The system is only asynchronous for a limited time and the other time is synchronous.
  + **Asynchronous:** There are no time limits of the messages.

**Failure Detectors:**

Are just algorithms to detect the failure of a node in the system. And the typical implementation is by sending a message to the node and if no response within a period of time then it will mark it as failed.

**Perfect Failure detector only exists in Synchronous, crash stop and reliable system model.**

**Broadcast**

**FIFO:** Messages sent by the same node must be delivered in the order they were sent.

**Causal broadcast:** ensures that messages are delivered in causal order: that is, if the broadcast of one message happened before the broadcast of another message, then all nodes must deliver those two messages in that order.

**Total Order Broadcast:** All nodes must deliver the messages in the same order.

It can be achieved by two approaches:

* **Single leader** approach sending all the message through a leader via FIFO broadcast, that gives total order broadcast out of the box. Limitation: the leader might fail and that will stop the broadcasting and changing it to a different leader is a tough task requires some protocols like Raft…
* **Lamport clocks:** to create a total order of events and they will be delivered in that order only t1 < t2 or t1 = t2 and N(1) < N(2) so e1 -> e2. The Limitation here is that a node can’t make sure that it receives all the messages. Given a defined set of events to all nodes that’s possible to total order them depending on the total order broadcast of Lamport clocks but as we deal with the unreliable network model, the messages might get delayed, and the nodes can’t deliver the messages if it’s not sure that all of the messages has arrived.

**Replicas:**

Keeps copy of the data on multiple machines, which will improve the query throughput. To deal with concurrent writes to the set of replicas we can use two approach. Clients will broadcast the operation write or read to all nodes:

* **Last write wins**: using Lamport clocks to total order the messages on the replicas but that might lead to data loss
* **Multi value register**: using vector clocks to keep both values in case the messages are concurrent.

To achieve fault tolerance in replicated settings we shouldn’t make the writes all the node and the read from all the nodes because we might face problems with messages got delayed or lost. So, we need some kind of consistency:

**Read after write consistency:**

We ensure that after a client writes a value, the same client will be able to read back the value it has just written. Using best effort replication.

After writing a client may not read the value it wrote because concurrently another client may have overwritten the value. Therefore, we say that read-after-write consistency requires reading either the last value written, or a later value. we could guarantee read-after write consistency by ensuring we always write to both replicas and/or read from both replicas. However, this would mean that reads and/or writes are no longer fault-tolerant: if one replica is unavailable, a write or read that requires responses from both replicas would not be able to complete.

To solve this problem, we use a:

**Quorum** : is a minimum set of nodes that must respond to some request for it to be successful.

In order to ensure read-after-write consistency, the quorum for the write and the quorum for the read must have a non-empty intersection: in other words, the read quorum must contain at least one node that has acknowledged the write.

**State Machine Replication**

Using FIFO Total order broadcast all the updated to all the replicas. So all the replicas will execute the operation in the same order and they will reach the same state. And the movement between state is deterministic. The limitation of the state machine replication are the limitations of total order broadcast. A node cannot deliver its message to itself before coordinating with the other nodes to all deliver it in the same order.

**primary-backup replication:**

The replication is done using a leader where all the update will happen on the leader and once they are committed the leader will FIFO broadcast them to the followers and they will commit them in the same order ensuring the total order broadcast properties.

**Using Causal broadcast in replication**

can happen but we need to make the changes commutative because the concurrent changes can’t be ordered unlike the updates that happened before each other, where the causal broadcast makes sure they will be delivered in an order that ensures causality.

**Replication using different kind of broadcast and their requirements:**

* **total order**: deterministic (SMR)
* **causal:** deterministic, concurrent updates commute
* **reliable:** deterministic, all updates commute
* **best-effort:** deterministic, commutative, idempotent, tolerates message loss

**2PL ensures serializable isolation for transactions, while 2PC ensures atomic commitment of distributed transactions.**

**Two Phase Commit Protocol**

It ensures the atomicity of transactions on multiple nodes to preserve the consistency of the system (all replicas have the same data). And since the coordinator is the single point of failure, then it is possible to avoid it by using a total order broadcast protocol.

The limitation of 2PC it is not enough for concurrent read and writes. We need a way to order the concurrent updates to make sure of the consistency between all the nodes.

**Serializability**

It means that concurrent transactions have the same effect as if they had been executed in some serial order. Does not give the order of the transactions.

**Linearizability**

Linearizability is a consistency model that ensures that all operations take effect atomically. It guarantees that when multiple clients concurrently access the data on multiple nodes the system will return the most up to date value. Linearizability provides a strong guarantee of consistency by ensuring that the system always appears to be in a consistent state, even in the presence of concurrent access.

So the nodes do not read outdated data.

**ABD algorithm:**

It ensures linearizable reads and writes, in the algorithm we assume that there is a write and read to a quorum. And it has a read after write consistency where the client will get the value and if it uses any different values in the returns then it will propagate the newer value to a replica, so this ensures that all read or write request do not finish until the node get acks from a quorum. And This will enforce linearizability.

**Raft**

Raft is a consensus algorithm that provides a reliable way to reach an agreement on the state of a distributed system in a fault-tolerant way ( by agreeing on a log which will lead to total order broadcast of all operations). It is designed to be easy to understand and is used for managing replicated state machines.

**assume a partially synchronous, crash-recovery system model.**

* **Leader Election:** The algorithm ensures that there is only one leader at a time.
* **Log Replication**: The leader replicates log entries to all other servers in the cluster.
* **Safety:** The algorithm ensures that if a server has applied a particular log entry to its state machine, then no other server will ever apply a different entry for the same log index.
* **Liveness:** The algorithm guarantees that all non-failing servers will eventually come to a consensus on the order of log entries.
* **Fault tolerance:** The algorithm can tolerate up to a certain number of server failures without compromising safety.

**Eventual Consistency**

The main problem with linearizability as a consistency model is that all operations required communication with a quorum of nodes like ABD and CAS algorithm to ensure linearizability. And for algorithms that use a leader to ensure the total order of the operation like Raft is somehow also limited by the leader capacity of handling requests on top of lots of communications over the network.

**The CAP Theorem:**

Consistency, Availability and Partition Tolerance. The systems must choose in the presence of a network partition to be consistent or available. In case of consistency, the node will be out of service because it will wait for a quorum response and not getting it due to the network partition or it can return its local value which might be an outdated value. But that increases the availability of the app for its users and results in better user experience for the app since the user can interact with the app even offline.

**Definition:**

It is a weaker model of consistency than linearizability. It does not need to wait for a quorum of nodes to answer to execute any operation.

Replicas process operations based only on their local state. If there are no more updates, eventually all replicas will be in the same state.

**Strong eventual consistency:**

* Eventual delivery: every update made to one non-faulty replica is eventually processed by every non-faulty replica.
* Convergence: any two replicas that have processed the same set of updates are in the same state (even if updates were processed in a different order).

**Properties:**

* Does not require waiting for network communication.
* Causal broadcast (or weaker) can disseminate updates.
* Concurrent updates generate potential conflicts that need to be resolved.

**CRDTs (Conflict free Replicated data types)**

**Operational Base**

**State Base**