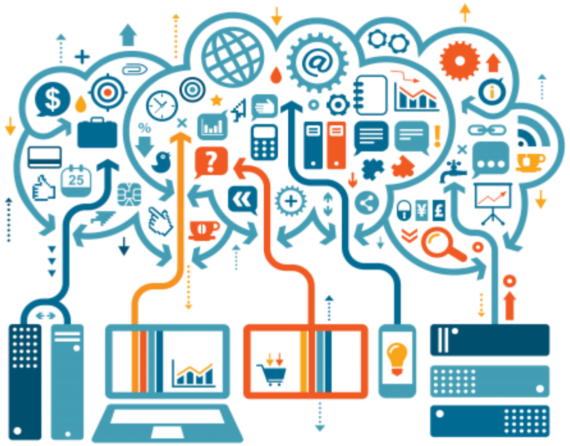
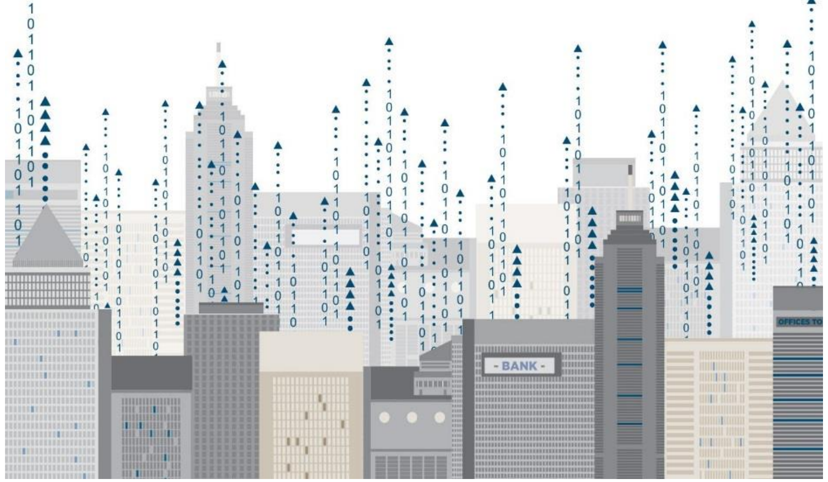




# Big Data Analytics

Fall 2025

## Lecture 1.5 – Basics of Distributed Computing


# Core Concepts

- Multiple Nodes (connected together through a network)
- Concurrency (all nodes work together at the same time)
- No shared clock (every node has its own clock)
- Independent Failures (any node can go down any time)
- Transparency (It seems to the user as if there is only one system serving only him and nothing is distributed)

# Core Features

- Scalability: you can add and remove nodes whenever you want – the system should not go down)
- Fault Tolerance: any node can go down any time, but the distributed system keeps on running
- Resource sharing: every node has its own set of hardware resources and there are software resources – all can be shared
- Performance: the goal is that a big query should not hang and results should come within a reasonable time (not in minutes) – many optimized queries finish in seconds

# Consensus: Paxos

# Paxos: Achieve Consensus

- Paxos is a **consensus algorithm** designed by Leslie Lamport (1998).
- It ensures that **multiple nodes in a distributed system agree on a single value**, even if some nodes fail or messages are delayed.
- It's widely used in databases, cloud platforms, and distributed systems for:
  - Leader election
  - Replicated state machines
  - Log replication (e.g., Google Chubby, Zookeeper, Cassandra, etcd).

# Core Idea

- Nodes (processes) must agree on **one value** out of possibly many proposed values.

- The algorithm ensures:
  - **Safety** – Only one value is chosen, and all nodes eventually learn it.
  - **Liveness** – If a majority of nodes are up and communicating, progress is made.

# Roles

- Each node can play multiple roles:
  1. **Proposer** – proposes a value.
  2. **Acceptor** – votes on proposals (majority needed).
  3. **Learner** – learns the final chosen value.

# The Algorithm (Phase 1)

- **Phase 1: Prepare / Promise**
- A Proposer chooses a proposal number *n* and sends a Prepare(*n*) request to a majority of Acceptors.
- Each Acceptor responds with a Promise not to accept proposals with a number less than n.
  - If the Acceptor has already accepted a proposal, it returns the highest-numbered proposal it has accepted.

# The Algorithm (Phase 2)

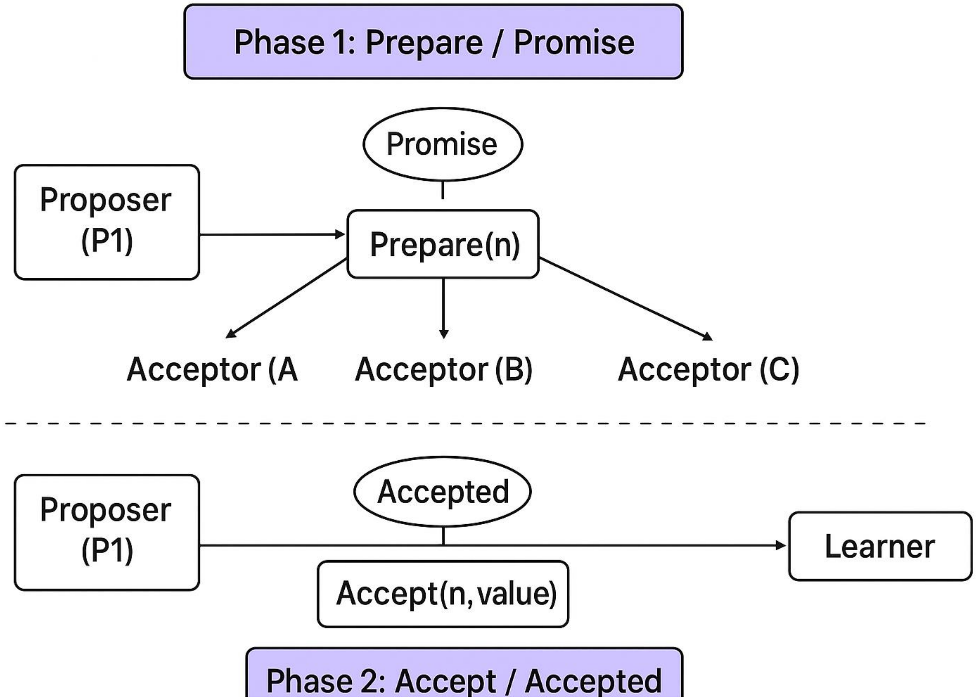
- **Phase 2: Accept / Accepted**
- If the Proposer receives a majority of **Promises**, it sends an **Accept(n, value)** request.
  - The value is either:
    - Its own proposed value, if no acceptor has accepted anything yet.
    - Otherwise, the value of the highest-numbered proposal returned by the Acceptors.
- Each Acceptor that receives **Accept(n, value)** accepts it, unless it has already promised to a higher-numbered proposal
- Once a value is accepted by a majority, it is **chosen**.
- **Learners** are informed of the chosen value.

# Example

- Proposer P1 sends Prepare(5) to Acceptors A, B, C.
- All reply with Promise (none have accepted before).
- P1 sends Accept(5, X) (proposing value X).
- A and B accept → majority reached → value X is chosen.
- If another proposer (P2) starts later with Prepare(7), it must respect already chosen values.

# Good and Bad things:

- Correctness: No two values are ever chosen.
- Fault tolerance: Works if a majority of nodes are alive.
- Drawback: Complex and can be slow due to multiple rounds.



# Consensus: RAFT

# Purpose

- Raft (by Diego Ongaro & John Ousterhout, 2014) is a **consensus algorithm** designed as a simpler alternative to Paxos.
- It ensures that a **cluster of servers** agree on a single consistent **log of operations**, even with failures.
- Used in **etcd, Consul, Kubernetes, RethinkDB, TiDB, CockroachDB**, and many other distributed systems.

# Core Idea

- Raft organizes consensus around a **leader** that manages log replication.
- At any time, one server is elected **Leader**.
- Other servers are **Followers**.
- Followers can become **Candidates** if the leader fails.
- Clients send requests to the Leader, which replicates them across servers.
- Agreement is achieved when the **majority of servers** confirm an entry.

# Roles

- Leader – handles client requests, manages replication.
- Follower – passive, just responds to requests from leader/candidate.
- Candidate – runs for leader election when no valid leader exists.

# Algorithm: Leader Election

- System starts with all servers as Followers.
- If a Follower doesn’t hear from a Leader within a timeout, it becomes a Candidate.
- The Candidate requests votes from others (RequestVote RPC).
- If it gets votes from the **majority**, it becomes the new Leader.
- If votes are split, a new election starts with a higher term.

# Algorithm: Log Replication

- Client sends a command → Leader appends it to its log.
- Leader sends AppendEntries RPC to Followers.
- Once a majority acknowledge the entry, the Leader marks it as committed.
- The Leader notifies Followers to also commit.

# Algorithm: Safety

- Raft ensures that only one leader per term exists.
- Followers only vote for candidates with logs at least as complete as their own (prevents conflicts).
- A committed log entry is guaranteed to persist across all future leaders.

# Algorithm: Log Compaction (Snapshots)

- Over time, logs grow large.
- Raft uses **snapshots** to compact the log.
- Old log entries are discarded once their effects are stored in a snapshot.

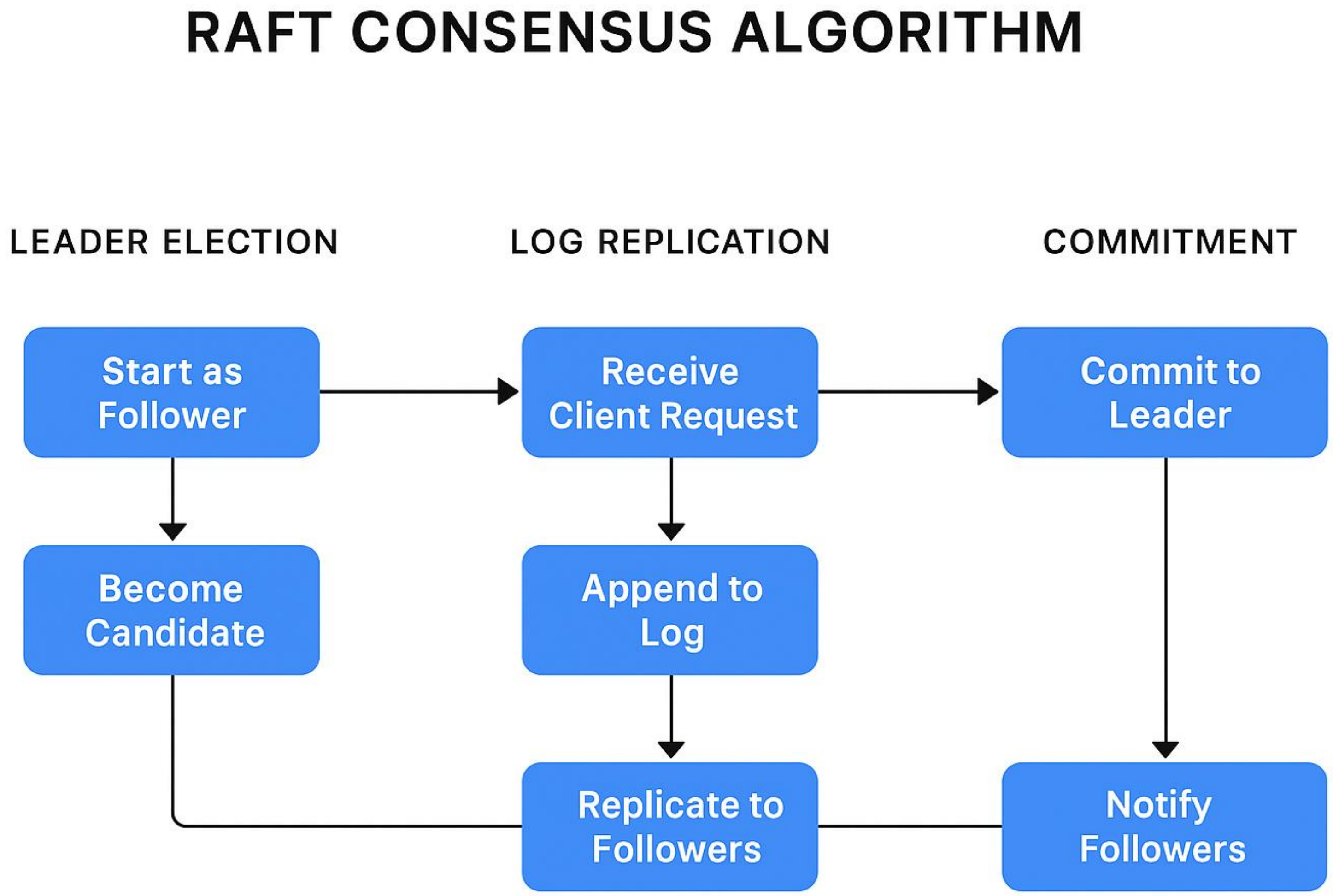
# Example

- Leader L1 receives a client request set x=5.
- L1 appends entry [x=5] to its log.
- L1 sends AppendEntries to Followers F1, F2.
- F1 and F2 acknowledge → majority achieved.
- L1 commits entry and notifies followers.
- All servers now have x=5 in their state.

# Properties

- Understandability: Raft was explicitly designed to be easier to teach and understand than Paxos.
- Safety: Ensures a log entry is never lost once committed.
- Liveness: Continues making progress as long as a majority of servers are available.
- Performance: Typically faster and easier to implement than Paxos in practice.

# Properties


RAFT CONSENSUS ALGORITHM
LEADER ELECTION
LOG REPLICATION
COMMITMENT
Start as Follower
Become Candidate
Receive Client Request
Append to Log
Replicate to Followers
Commit to Leader
Notify Followers

# Synchronization: Lamport Timestmp

# Properties

- In a **distributed system**, there’s no global clock. Events happen on different machines, and we need a way to **order them consistently**.
- Leslie Lamport (1978) introduced **logical clocks** to assign timestamps to events so we can determine **happens-before (causal ordering)**.

# Happens Before →

- A → B if
- Both are events in the same process and A occurs before B
- A is the sending and B is the receiving
- Transitive Closure: If A→B and B→C then A→C

# Rules

- Each process Pi maintains a logical clock Li (an integer counter).
- Increment before each event:
  - Before executing an event, process Pi does: $L_i = L_i + 1$
- Message Sending Rule:
  - When a process sends a message m, it attaches its timestamp: $timestamp(m) = L_i$
- Message Receiving Rule:
  - When a process Pj receives message m with timestamp $T_m$: $L_j = \max(L_j, T_m) + 1$
  - This ensures that the receiver's clock is always ahead of (or equal to) the sender's clock.

# Example

- P1 and P2
- Initial clocks: L1=0 and L2=0
- P1 does an event: L1=1
- P1 sends m – attaches timestamp 1
- P2 receives m: L2=max(L2,1)+1 = 2
- Consistent now: send event(1) and receive event(2)

# Properties

- Provides partial ordering
- A→B then timestamp(a) < timestamp(b)
- But if timestamp(a) < timestamp(b), then no guarantee that A→B

# Synchronization: Vector Clocks

# Core Idea

- **Vector clocks** improve on Lamport by capturing **causal relationships** more precisely, allowing us to detect **concurrent events**.
- Each process keeps a vector of logical clocks (an array of integers).
- If there are N processes, then each process Pi maintains a vector Vi of length N
- Vi[j] = process Pi’s knowledge of process Pj’s logical time.

# Rules

- Initialization
  - All entries start at 0.
  - For process Pi: Vi[i] = 0.
- Local Event at Pi
  - Increment its own entry
  - Vi[i] = Vi[i] + 1
- Message sending by Pi
  - Increment its own entry
  - Vi[i] = Vi[i] + 1
- Message receiving by Pj
  - On receiving message with vector Vm:
  - For each k: Vj[k] = max(Vj[k], Vm[k])
  - Vj[j] = Vj[j] + 1

# Comparing timestamps

- Events a and b have vector timestamps Va and Vb
- a→b if Va[i] ≤ Vb[i] for all i, and Va[j] < Vb[j] for at least one j.
- A and b are concurrent events if neither Va ≤ Vb nor Vb ≤ Va.

# Example

- V1 = [0,0,0], V2 = [0,0,0], V3 = [0,0,0]
- V1 = [1,0,0] (P1 does an event)
- V1 = [2,0,0], message carries [2,0,0] (P1 messages P2)
- P2 receives:
  - V2 = max([0,0,0], [2,0,0]) = [2,0,0]
  - V2[2] = V2[2] + 1 → [2,1,0]

# Properties

- Causal ordering: Precisely identifies if one event happened before another.
- Concurrency detection: If vectors are incomparable, events are concurrent.
- Overhead: Requires N integers per process (scales with number of processes).
- Summary:
  - Lamport: Partial ordering
  - Vector: full causal ordering + detects concurrency
- Distributed debugging, version control, DynamoDB, Cassandra