Big Data Analytics: Lecture 1.5 – Basics of Distributed Computing

Dr. Tariq Mahmood

Fall 2025

1 Core Concepts of Distributed Computing

A distributed system is a collection of independent computers that appears to its users as a single coherent system. These systems are essential for modern applications that require high availability, fault tolerance, and scalability. The core concepts that define a distributed system are:

- **Multiple Nodes:** A distributed system consists of multiple computers, referred to as nodes, that are connected through a network. These nodes work together to achieve a common goal by communicating and coordinating their actions.
- **Concurrency:** The components of a distributed system run simultaneously, allowing for parallel processing and handling of multiple requests at the same time.
- **No Shared Clock:** In a distributed system, there is no global clock that all nodes can rely on for synchronization. Each node has its own clock, which can lead to challenges in ordering events across the system.
- **Independent Failures:** Any node in a distributed system can fail independently without causing the entire system to fail. This fault tolerance is a key advantage of distributed systems.
- **Transparency:** A distributed system should hide its complexity from the user, making it seem as if there is only one system serving them. This includes hiding the distribution of data and processes across multiple nodes.

1.1 Core Features

- **Scalability:** Distributed systems are designed to be scalable, meaning you can add or remove nodes as needed without bringing the system down. This allows the system to handle increasing workloads.
- **Fault Tolerance:** If a node fails, the distributed system can continue to operate, ensuring high availability. This is often achieved through redundancy and replication of data and services.
- **Resource Sharing:** In a distributed system, both hardware and software resources can be shared among the different nodes.
- **Performance:** The goal is to provide results for large queries in a reasonable amount of time, often within seconds. This is achieved by distributing the workload across multiple nodes and processing it in parallel.

2 Consensus: Paxos

Paxos is a consensus algorithm created by Leslie Lamport that allows a group of computers in a distributed system to agree on a single value, even in the presence of failures. It is widely used in systems where high reliability and fault tolerance are crucial.

2.1 Core Idea

The main goal of Paxos is for nodes (processes) to agree on one value from a set of proposed values. The algorithm guarantees the following:

- **Safety:** Only one value is ever chosen, and once chosen, all nodes will eventually learn it.
- **Liveness:** If a majority of the nodes are operational and can communicate with each other, the system will continue to make progress.

2.2 Roles

In the Paxos algorithm, each node can take on one or more of the following roles:

- **Proposer:** Proposes a value to be agreed upon.
- **Acceptor:** Votes on the proposals. A majority of acceptors is needed to reach a consensus.
- **Learner:** Learns the final chosen value once a consensus is reached.

2.3 The Algorithm

The Paxos algorithm operates in two phases:

2.3.1 Phase 1: Prepare / Promise

1. A Proposer selects a unique proposal number, n, and sends a *Prepare(n)* request to a majority of the Acceptors.
2. Each Acceptor that receives the *Prepare(n)* request responds with a *Promise* not to accept any proposals with a number less than n. If the Acceptor has already accepted a proposal, it includes the highest-numbered proposal it has accepted in its response.

2.3.2 Phase 2: Accept / Accepted

1. If the Proposer receives a *Promise* from a majority of the Acceptors, it sends an *Accept(n, value)* request to those Acceptors. The value is either its own proposed value (if no other value was returned by the Acceptors) or the value of the highest-numbered proposal returned by the Acceptors.
2. An Acceptor that receives the *Accept(n, value)* request accepts it, unless it has already promised to consider only proposals with a number higher than n.
3. Once a value is accepted by a majority of the Acceptors, that value is chosen.
4. Learners are then informed of the chosen value.

2.4 Worked Example

Let's consider a scenario with one Proposer (P1) and three Acceptors (A, B, C).

1. Proposer P1 wants to propose a value X. It chooses a proposal number, let's say 5, and sends a *Prepare(5)* request to Acceptors A, B, and C.
2. Since none of the Acceptors have accepted any proposal before, they all reply with a *Promise* to not accept any proposal with a number less than 5.
3. P1 has now received a majority of Promises (in this case, from all three Acceptors). It then sends an *Accept(5, X)* request to A, B, and C.
4. Let's say Acceptors A and B receive the request and accept it. At this point, a majority (2 out of 3) has been reached.
5. The value X is now chosen. Even if another Proposer (P2) later starts with a higher proposal number, say 7, by sending a *Prepare(7)* request, the Acceptors that have already accepted X will inform P2 about the accepted value, and P2 will have to propose the same value X.

2.5 Advantages and Disadvantages

- **Advantages:**
  - **Correctness:** Guarantees that no two different values will ever be chosen.
  - **Fault tolerance:** The system continues to work as long as a majority of the nodes are operational.
- **Disadvantages:**
  - **Complexity:** The Paxos algorithm is known to be complex and difficult to implement correctly.
  - **Performance:** The multiple rounds of communication can make it slow in some scenarios.

3 Consensus: RAFT

Raft is a consensus algorithm designed by Diego Ongaro and John Ousterhout in 2014 as a more understandable alternative to Paxos. It is widely used in many distributed systems like etcd, Consul, and Kubernetes.

3.1 Core Idea

Raft organizes the consensus process around a single leader.

- At any given time, one server is elected as the **Leader**.
- All other servers are **Followers**.
- If the leader fails, a Follower can become a **Candidate** to start a new election.
- All client requests are sent to the Leader, which then replicates them to the Followers.
- A consensus is reached when a majority of the servers confirm an entry.

3.2 Roles

- **Leader:** Handles all client requests and manages log replication.
- **Follower:** Is passive and only responds to requests from the Leader or Candidates.
- **Candidate:** A Follower that starts an election to become the new Leader if the current Leader is not available.

3.3 The Algorithm

3.3.1 Leader Election

1. The system starts with all servers as Followers.
2. If a Follower doesn't receive a heartbeat from a Leader within a certain timeout period, it becomes a Candidate.
3. The Candidate requests votes from the other servers.
4. If the Candidate gets votes from a majority of the servers, it becomes the new Leader.
5. If the votes are split and no candidate gets a majority, a new election starts with a higher term number.

3.3.2 Log Replication

1. A client sends a command to the Leader. The Leader appends this command to its log.
2. The Leader then sends an *AppendEntries* RPC (Remote Procedure Call) to the Followers.
3. Once a majority of the Followers acknowledge the entry, the Leader marks it as committed.
4. The Leader then notifies the Followers to also commit the entry.

3.3.3 Safety

Raft includes safety mechanisms to ensure correctness:

- Only one leader can exist in a given term.
- Followers will only vote for a candidate if that candidate's log is at least as up-to-date as their own, which prevents conflicts.
- A committed log entry is guaranteed to be preserved across all future leaders.

3.3.4 Log Compaction (Snapshots)

- To prevent logs from growing indefinitely, Raft uses snapshots.
- Once the state of the system is saved in a snapshot, the older log entries can be discarded.

3.4 Worked Example

Imagine a distributed key-value store with a Leader (L1) and two Followers (F1, F2).

1. A client sends a request to Leader L1 to set the value of x to 5.
2. L1 appends the entry '[x=5]' to its log.
3. L1 sends an *AppendEntries* RPC with this entry to Followers F1 and F2.
4. Both F1 and F2 receive the entry, append it to their logs, and send an acknowledgment back to L1.
5. L1 has now received acknowledgments from a majority of the servers (itself, F1, and F2). It commits the entry.
6. L1 then notifies F1 and F2 that the entry is committed.
7. All servers now have the state x=5.

3.5 Properties and Advantages

- **Understandability:** Raft was designed to be easier to understand and implement than Paxos.
- **Safety:** It guarantees that once a log entry is committed, it will never be lost.
- **Liveness:** The system continues to make progress as long as a majority of servers are available.
- **Performance:** In practice, Raft is often faster and easier to implement than Paxos.

4 Synchronization: Lamport Timestamps

In a distributed system, there's no global clock to order events. Leslie Lamport introduced the concept of logical clocks in 1978 to assign timestamps to events, allowing us to determine a causal ordering known as the "happens-before" relationship.

4.1 Happens Before (→)

The "happens-before" relationship, denoted by →, is defined as follows:

- If A and B are events in the same process and A occurs before B, then A → B.
- If A is the sending of a message by one process and B is the receipt of that message by another process, then A → B.
- If A → B and B → C, then A → C (transitive closure).

4.2 Rules

1. Each process Pi maintains a logical clock, Li, which is an integer counter.
2. **Increment before each event:** Before executing an event, process Pi increments its clock: $Li = Li + 1$.
3. **Message Sending Rule:** When a process sends a message m, it includes its timestamp: $timestamp(m) = Li$.
4. **Message Receiving Rule:** When a process Pj receives a message m with timestamp Tm, it updates its clock: $Lj = \max(Lj, Tm) + 1$. This ensures the receiver's clock is always ahead of the sender's.

4.3 Worked Example

Consider two processes, P1 and P2.

1. Initial clocks: L1 = 0, L2 = 0.
2. P1 has an internal event. It increments its clock: L1 = 1.
3. P1 sends a message m to P2. The message carries the timestamp 1.
4. P2 receives the message m. It updates its clock: L2 = max(L2, 1) + 1 = max(0, 1) + 1 = 2.
5. The events are now consistently ordered: the send event has a timestamp of 1, and the receive event has a timestamp of 2.

4.4 Properties and Disadvantages

- **Provides partial ordering:** If A → B, then timestamp(a) ¡ timestamp(b).
- **Limitation:** If timestamp(a) ¡ timestamp(b), it does not necessarily mean that A → B. The events could be concurrent. Lamport timestamps cannot distinguish between causally related and concurrent events.

5 Synchronization: Vector Clocks

Vector clocks improve upon Lamport timestamps by capturing causal relationships more accurately, which allows for the detection of concurrent events.

5.1 Core Idea

- Each process keeps a vector of logical clocks, which is an array of integers.
- If there are N processes in the system, each process Pi maintains a vector Vi of length N.
- The entry Vi[j] represents Pi's knowledge of the logical time of process Pj.

5.2 Rules

1. **Initialization:** All entries in all vectors start at 0.
2. **Local Event at Pi:** Increment its own entry in its vector: $Vi[i] = Vi[i] + 1$.
3. **Message Sending by Pi:** First, increment its own entry: $Vi[i] = Vi[i] + 1$. Then, send the entire vector Vi with the message.
4. **Message Receiving by Pj:** On receiving a message with vector Vm:
   - For each element k, update its vector: $Vj[k] = \max(Vj[k], Vm[k])$.
   - Increment its own entry: $Vj[j] = Vj[j] + 1$.

5.3 Comparing Timestamps

Given two events a and b with vector timestamps Va and Vb:

- a $\rightarrow$ b if Va[i] $\leq$ Vb[i] for all i, and Va[j] ¡ Vb[j] for at least one j.
- a and b are concurrent if neither Va $\leq$ Vb nor Vb $\leq$ Va.

5.4 Worked Example

Consider three processes P1, P2, and P3.

1. Initial vectors: V1 = [0,0,0], V2 = [0,0,0], V3 = [0,0,0].
2. P1 has a local event. It increments its own clock: V1 = [1,0,0].
3. P1 sends a message to P2. First, it increments its clock: V1 = [2,0,0]. The message carries the vector [2,0,0].
4. P2 receives the message. It updates its vector by taking the element-wise maximum of its own vector and the received vector: V2 = max([0,0,0], [2,0,0]) = [2,0,0].
5. P2 then increments its own entry: The second element of V2 becomes V2[1] + 1, but the slide uses 1-based indexing for the process number and 0-based for the vector. Let's assume the slide means the second entry for P2. So, V2 = [2, 1, 0].

5.5 Properties and Applications

- **Causal ordering:** Precisely identifies if one event happened before another.
- **Concurrency detection:** If the vectors are incomparable, the events are concurrent.
- **Overhead:** Requires N integers per process, which can be a significant overhead as the number of processes increases.
- **Summary:** While Lamport clocks provide partial ordering, vector clocks provide full causal ordering and can detect concurrency.
- **Applications:** Used in distributed debugging, version control systems, and in databases like DynamoDB and Cassandra.