# Consensus in Distributed Systems

## From Impossibility to Paxos

Dr. Elena Vasquez

Department of Computer Science
Meridian University

Fall 2025

# Lecture Overview

# The Consensus Problem

A set of $n$ distributed processes must agree on a single output value from a set of proposed input values, even in the presence of failures.

# The Consensus Problem

A set of $n$ distributed processes must agree on a single output value from a set of proposed input values, even in the presence of failures.

**Required Properties:**

- **Agreement:** Every correct process outputs the same value.

# The Consensus Problem

A set of $n$ distributed processes must agree on a single output value from a set of proposed input values, even in the presence of failures.

**Required Properties:**

- **Agreement:** Every correct process outputs the same value.
- **Termination:** Every correct process eventually decides.

# The Consensus Problem

A set of $n$ distributed processes must agree on a single output value from a set of proposed input values, even in the presence of failures.

**Required Properties:**

- **Agreement:** Every correct process outputs the same value.
- **Termination:** Every correct process eventually decides.
- **Validity:** The output value was proposed by some process.

# The Consensus Problem

A set of $n$ distributed processes must agree on a single output value from a set of proposed input values, even in the presence of failures.

**Required Properties:**

- **Agreement:** Every correct process outputs the same value.
- **Termination:** Every correct process eventually decides.
- **Validity:** The output value was proposed by some process.
- **Integrity:** Each process decides at most once.

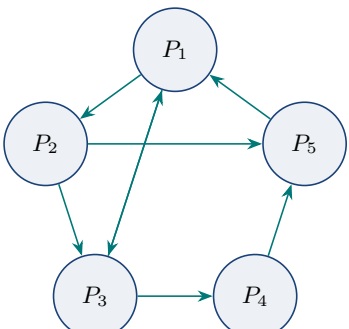
# Why Consensus Matters

**Real-world applications:**

- **Leader Election:** Selecting a coordinator among a group of servers. Used in NimbusDB for automatic failover across data centers.
- **State Machine Replication:** Keeping multiple replicas consistent. The Apex Cloud platform uses replicated state machines for its configuration service.
- **Atomic Broadcast:** Ensuring all nodes receive messages in the same order. Vertex's trading engine depends on total-order broadcast for audit trails.

Consensus is the foundational primitive that enables fault-tolerant distributed coordination.

# System Model




**Assumptions:**

- Asynchronous message passing — no bounds on message delay.
- Processes communicate via reliable channels.
- Crash-stop failures: a failed process stops forever.
- No Byzantine (arbitrary) failures.
- At most $f < n/2$ processes may fail.

# FLP Impossibility Result

In a fully asynchronous distributed system, it is impossible to achieve deterministic consensus in the presence of even a single crash failure.

# FLP Impossibility Result

In a fully asynchronous distributed system, it is impossible to achieve deterministic consensus in the presence of even a single crash failure.

**What this means:**

- No deterministic protocol can guarantee both **safety** (agreement) and **liveness** (termination) under full asynchrony.

# FLP Impossibility Result

In a fully asynchronous distributed system, it is impossible to achieve deterministic consensus in the presence of even a single crash failure.

**What this means:**

- No deterministic protocol can guarantee both **safety** (agreement) and **liveness** (termination) under full asynchrony.
- The adversary can delay messages arbitrarily, preventing a process from distinguishing a slow process from a crashed one.

**Implication:** Practical systems must either assume partial synchrony or use randomization to

# FLP Impossibility Result

In a fully asynchronous distributed system, it is impossible to achieve deterministic consensus in the presence of even a single crash failure.

**What this means:**

- No deterministic protocol can guarantee both **safety** (agreement) and **liveness** (termination) under full asynchrony.
- The adversary can delay messages arbitrarily, preventing a process from distinguishing a slow process from a crashed one.
- This is a **fundamental result** — not an engineering limitation but a theoretical impossibility.

**Implication:** Practical systems must either assume partial synchrony or use randomization to

# Paxos: Core Ideas

**Developed by Leslie Lamport (1989).** Named after the legislative process on the Greek island of Paxos.

**Three Roles:**

- **Proposers:** Initiate consensus by proposing values.
- **Acceptors:** Vote on proposals; a *majority quorum* must agree.
- **Learners:** Learn the chosen value once consensus is reached.

**Key Insight:** A process can play multiple roles simultaneously.

If any value is chosen, every subsequent chosen value must be identical. Paxos guarantees this even with $f < n/2$ failures.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

1. **Proposer** selects a unique, monotonically increasing proposal number $n$.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

1. **Proposer** selects a unique, monotonically increasing proposal number $n$.
2. Sends $\textsc{Prepare}(n)$ to a **majority** of acceptors.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

1. **Proposer** selects a unique, monotonically increasing proposal number $n$.
2. Sends $\textsc{Prepare}(n)$ to a **majority** of acceptors.
3. Each **Acceptor** that receives $\textsc{Prepare}(n)$:
   - If $n$ is the highest it has seen, it **promises** to reject proposals with lower numbers.
   - It replies with $\textsc{Promise}(n, v, n')$ where $v$ is the highest-numbered proposal it has already accepted (with number $n' < n$), or $\perp$ if none.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

1. **Proposer** selects a unique, monotonically increasing proposal number $n$.
2. Sends $\textsc{Prepare}(n)$ to a **majority** of acceptors.
3. Each **Acceptor** that receives $\textsc{Prepare}(n)$:
   - If $n$ is the highest it has seen, it **promises** to reject proposals with lower numbers.
   - It replies with $\textsc{Promise}(n, v, n')$ where $v$ is the highest-numbered proposal it has already accepted (with number $n' < n$), or $\perp$ if none.
4. The proposer collects promises. If it hears from a majority, Phase 2 begins.

# Paxos: Phase 1 — Prepare & Promise

**Goal:** Elect a leader and learn about any previously chosen values.

1. **Proposer** selects a unique, monotonically increasing proposal number $n$.
2. Sends $\textsc{Prepare}(n)$ to a **majority** of acceptors.
3. Each **Acceptor** that receives $\textsc{Prepare}(n)$:
   - If $n$ is the highest it has seen, it **promises** to reject proposals with lower numbers.
   - It replies with $\textsc{Promise}(n, v, n')$ where $v$ is the highest-numbered proposal it has already accepted (with number $n' < n$), or $\perp$ if none.
4. The proposer collects promises. If it hears from a majority, Phase 2 begins.

An acceptor always remembers the highest proposal number it has promised and the highest-numbered proposal it has accepted.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.
2. Proposer sends $\textsc{Accept}(n, v)$ to the same majority.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.
2. Proposer sends $\textsc{Accept}(n, v)$ to the same majority.
3. Each **Acceptor** accepts $(n, v)$ *only if* it has not promised a proposal $> n$.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.
2. Proposer sends $\textsc{Accept}(n, v)$ to the same majority.
3. Each **Acceptor** accepts $(n, v)$ *only if* it has not promised a proposal $> n$.
4. When a majority of acceptors have accepted $(n, v)$, the value $v$ is **chosen**.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.
2. Proposer sends $\textsc{Accept}(n, v)$ to the same majority.
3. Each **Acceptor** accepts $(n, v)$ *only if* it has not promised a proposal $> n$.
4. When a majority of acceptors have accepted $(n, v)$, the value $v$ is **chosen**.
5. **Learners** are notified of the chosen value.

# Paxos: Phase 2 — Accept & Learn

**Goal:** Drive the system to choose a single value.

1. The **Proposer** examines the Promise replies:
   - If any acceptor returned a previously accepted value, the proposer **must** use the value with the highest $n'$.
   - Otherwise, it is free to propose its own value $v$.
2. Proposer sends $\textsc{Accept}(n, v)$ to the same majority.
3. Each **Acceptor** accepts $(n, v)$ *only if* it has not promised a proposal $> n$.
4. When a majority of acceptors have accepted $(n, v)$, the value $v$ is **chosen**.
5. **Learners** are notified of the chosen value.

A value is *chosen* once a majority of acceptors have accepted it for the same proposal number. All future decisions will agree on this value.

# Summary & Key Takeaways

1. **Consensus** is the problem of getting distributed processes to agree on a single value despite failures.
2. The **FLP impossibility** result proves that deterministic consensus is impossible in a purely asynchronous model with even one crash.
3. **Paxos** provides a practical solution by assuming eventual synchrony (or using randomization), achieving safety under all conditions.
4. The two-phase structure — **Prepare/Promise** followed by **Accept/Learn** — is the core mechanism.
5. Modern systems (NimbusDB, Apex Cloud, Vertex) build on these primitives for fault-tolerant coordination.

## Questions?