Production and Operations Management

INFT 3611

Dr. J. Burns

November 30, 2025

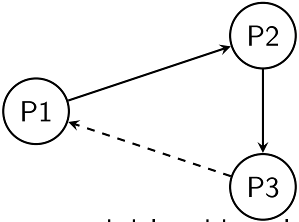
Section 2: Architectures

This content is based on the following public resources: https://www.distributed-systems.net/index.php/books/ds4/

Distributed Computation is Hard

Problem

Distributed systems consist of independently executing processes that communicate over unreliable networks. Each process has only a partial, delayed, and inconsistent view of global state.



Different and delayed knowledge

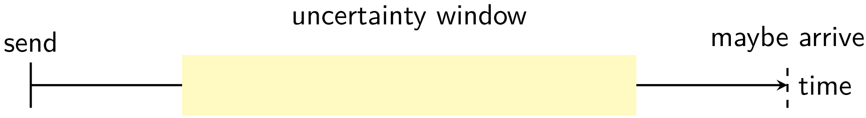
Distributed Computation — Solutions

- **Logical clocks** for ordering without synchronized time.
- **Quorum-based decisions** instead of global agreement.
- **Replicated state machines** for deterministic consistency.
- **Retry and reconciliation** to overcome uncertainty.
- **Failure detectors** to approximate liveness information.

Asynchrony

Problem

Message delays are unbounded. A node cannot distinguish a slow peer from a failed or partitioned one. Silence has no meaning.



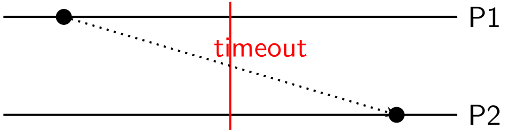
Asynchrony — Solutions

- **Partial synchrony assumptions** for eventual timing bounds.
- **Randomized timeouts** to avoid synchronized failure.
- **Quorum reads/writes** to avoid waiting for all nodes.
- **Eventual failure detectors** that improve over time.
- **Logical ordering** instead of timing-based decisions.

Timeout Ambiguity

Problem

A timeout indicates missing messages, not a crash. Reacting prematurely can cause split-brain or two leaders if delayed messages later arrive.



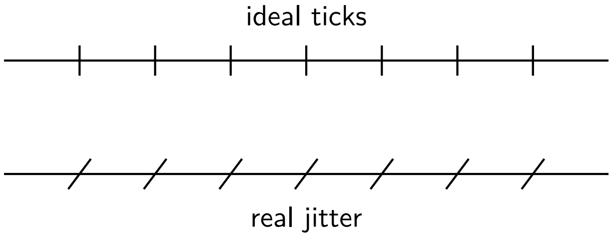
Timeout Ambiguity — Solutions

- **Terms/epochs** invalidate outdated leaders (Raft).
- **Leases** provide time-bounded authority.
- **Commit rules** require quorum confirmation.
- **Redundant communication** to detect restored nodes.
- **Delayed leadership transition** to avoid flapping.

Synchronous Model

Problem

The synchronous model assumes predictable message and processing times. Real systems rarely meet these assumptions, causing synchronous algorithms to fail under modest delay.



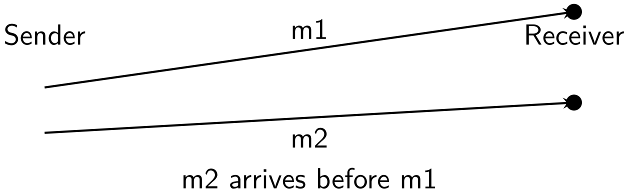
Synchronous Model — Solutions

- **Treat synchrony as conceptual only**.
- **Design protocols for weaker models** (e.g., Raft, Paxos).
- **Avoid timing-based correctness assumptions**.
- **Use timeouts as hints, not proofs of failure**.
- **Use logical clocks** when real time is unreliable.

Message Reordering

Problem

Messages may traverse different network paths or face variable queueing delays. As a result, they can arrive in an order different from the one in which they were sent.



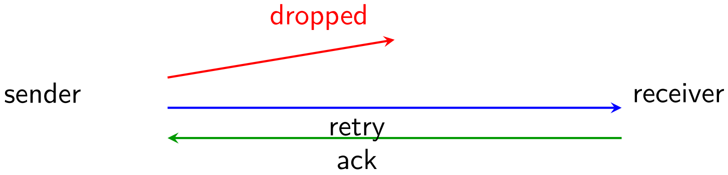
Message Reordering — Solutions

- **Sequence numbers** to restore intended order.
- **Version vectors** to track causality across replicas.
- **Idempotent operations** to tolerate duplicates.
- **Deterministic replay** in replicated state machines.
- **Ordering layers** such as TCP or application-level queues.

Message Loss

Problem

Networks drop messages due to congestion, queue overflow, or transient link faults. Loss prevents updates from propagating and may leave replicas inconsistent.



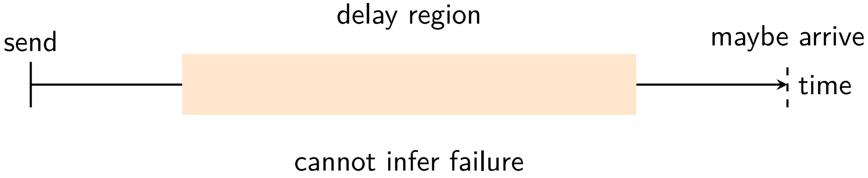
Message Loss — Solutions

- **Retransmissions** until confirmation is received.
- **Acknowledgments** to confirm delivery.
- **Checksums** to detect corruption.
- **Backoff algorithms** to reduce congestion.
- **Periodic anti-entropy** to reconcile state.

Arbitrary Delay

Problem

Messages may be delayed indefinitely. Such delays look identical to failures, making timing unreliable for determining remote state or coordinating decisions.



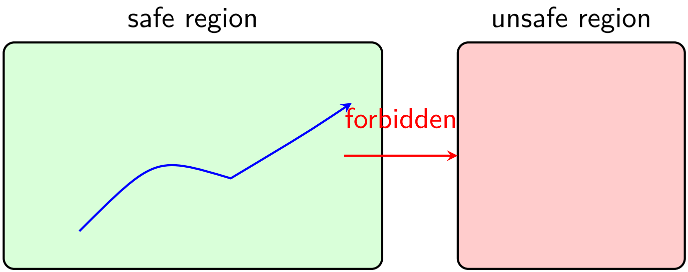
Arbitrary Delay — Solutions

- **Quorum-based operations** to tolerate missing responses.
- **Randomized timeouts** to reduce coordination conflicts.
- **Eventually perfect failure detectors** improving over time.
- **Epoch-based leadership** to avoid split-brain.
- **Delay-independent safety rules** in consensus protocols.

Safety

Problem

A system violates safety if it reaches an invalid state—such as electing two leaders or committing conflicting operations. Safety violations cannot be undone once they occur.



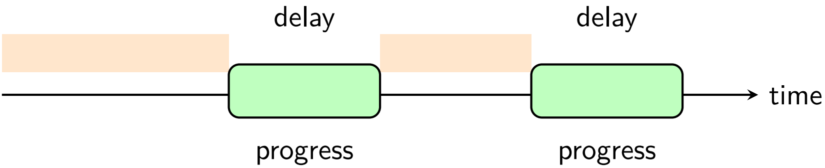
Safety — Solutions

- **Single-writer leadership** to serialize decisions.
- **State machine replication** for deterministic behavior.
- **Quorum-based commits** to avoid conflicting decisions.
- **Monotonic state transitions** (legal state space).
- **Invariant-preserving protocols** such as Raft/Paxos.

Liveness

Problem

Even if safety holds, a system may stall indefinitely under heavy delay or unstable leadership. Liveness requires eventual forward progress.



Liveness — Solutions

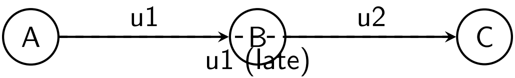
- **Randomized leader election** to break symmetry.
- **Eventually stable timeouts** after network recovery.
- **Retry loops** to ensure completion under partial failure.
- **Failure detectors** that improve accuracy over time.
- **Progress conditions** built into consensus protocols.

Algorithm 1 Retry-with-Ack Protocol

```
send(msg)
while no ack received do
    wait(timeout)
    resend(msg)
end while
```

Problem

Replicas receive updates in different orders and at different times. Without coordination, their states may temporarily diverge, producing inconsistent reads across the system.



temporary divergence

Eventual Consistency — Solutions

- **Anti-entropy protocols** to exchange and reconcile state.
- **Version vectors** for detecting missing causal updates.
- **Idempotent updates** to tolerate replay and duplicates.
- **Convergent conflict-resolution rules** (e.g., LWW).
- **Periodic gossip** for reliable dissemination.

Eventual Consistency — Anti-Entropy Algorithm

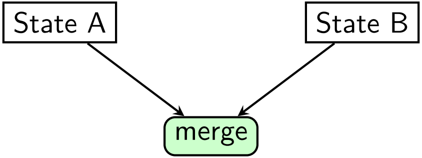
Algorithm 2 Periodic Anti-Entropy

```
while true do
    wait(random_interval())
    peer ← pickRandomReplica()
    send(localState, peer)
    remoteState ← receive(peer)
    localState ← merge(localState, remoteState)
end while
```

CRDT Convergence

Problem

Conflict-free Replicated Data Types require carefully designed merge rules. Incorrect merge semantics or non-commutative operations can cause replicas to diverge permanently.



deterministic, convergent result

CRDT Convergence — Solutions

- **Commutative updates** so order does not matter.
- **Associative merge operations** for consistent folding.
- **Idempotent merges** to handle duplicate state.
- **State-based CRDTs** for robust reconciliation.
- **Grow-only or monotonic structures** to avoid conflicts.

CRDT — Version Vector Update Algorithm

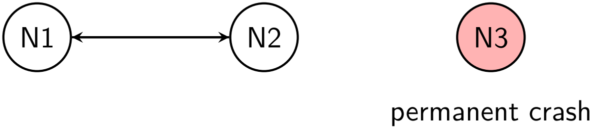
Algorithm 3 Version Vector Update

```
vv[node] ← vv[node] + 1
attach(operation, vv)
broadcast(operation)
onReceive(op):
    vv[op.src] ← max(vv[op.src], op.vv[op.src])
    apply(op)
```

Crash-Stop Failures

Problem

Nodes may halt permanently due to hardware faults or power loss. They never recover or rejoin, reducing available replicas and risking loss of majority quorums.



Crash-Stop — Solutions

- **Majority quorums** to tolerate permanent minority loss.
- **Replicated logs** to keep state despite node loss.
- **Leader re-election** to maintain progress.
- **Durable writes** that survive local crashes.
- **Static membership** for predictable fault tolerance.

Crash-Recovery Log Reconstruction

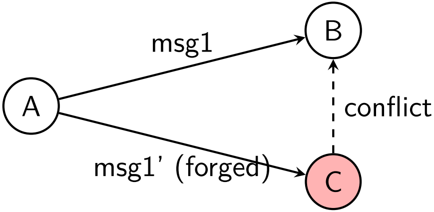
Algorithm 4 Crash-Recovery Log Rebuild

```
onStartup():
    log ← loadFromDisk()
    lastIndex ← log.tail
    send(hello, peers)
    missing ← fetchEntries(lastIndex, peers)
    append(log, missing)
    restoreState(log)
```

Byzantine Failures

Problem

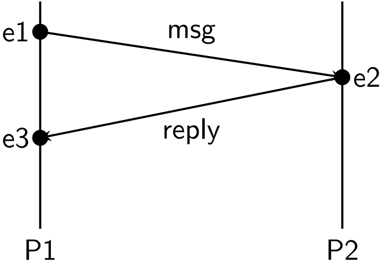
Nodes may behave arbitrarily: forging messages, equivocating, or selectively dropping updates. This breaks assumptions in non-Byzantine protocols and can mislead peers.



Lamport Clocks

Problem

Without synchronized clocks, processes cannot agree on the order of events using physical time. Concurrent events make real-time ordering ambiguous.



- **Monotonic counters** incremented on each event.
- **Piggyback timestamps** on every message.
- **Max-merge rule** to update local clocks.
- **Defines happens-before** without physical time.
- **Total order** via tie-breaking with process ID.

Lamport Clock Update Algorithm

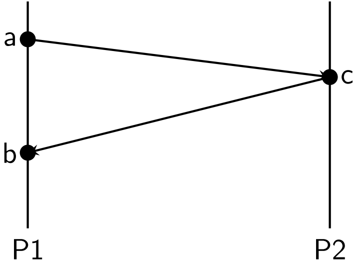
Algorithm 5 Lamport Timestamp Update

```
onLocalEvent():
    clock ← clock + 1
onSend(msg):
    clock ← clock + 1
    msg.ts ← clock
    send(msg)
onReceive(msg):
    clock ← max(clock, msg.ts) + 1
```

Happens-Before Relation

Problem

Determining causality between events is difficult without a global timeline. Concurrent events appear unordered in real time.



Happens-Before — Solutions

- **Lamport timestamps** encode causal precedence.
- **HB defines concurrency** when neither event dominates.
- **Prevents reordering errors** in replicated logs.
- **Supports deterministic replay**.
- **Enables causal consistency** across replicas.

Vector Clocks

Problem

Lamport clocks cannot distinguish true concurrency from causality. Systems require richer metadata to track causality accurately.

$$\boxed{[2,1,0]} \xrightarrow{\text{dominates}} \boxed{[2,1,3]}$$

componentwise comparison

Vector Clocks — Solutions

- **Track causality** per process dimension.
- **Detect concurrency** using vector comparison rules.
- **Support multi-writer replication**.
- **Drive conflict resolution** in CRDTs.
- **Minimal metadata** for consistent ordering.

Vector Clock Update Algorithm

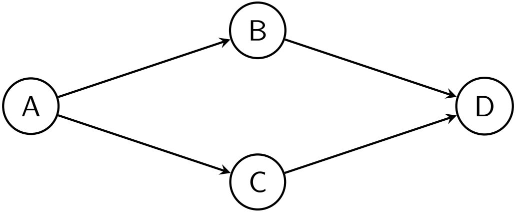
Algorithm 6 Vector Clock Update

```
onLocalEvent():
    VC[i] ← VC[i] + 1
onSend(op):
    VC[i] ← VC[i] + 1
    op.VC ← VC
onReceive(op):

for each process j do
        VC[j] ← max(VC[j], op.VC[j])
end for
    VC[i] ← VC[i] + 1
```

Problem

Broadcasting updates reliably in large clusters is expensive. Loss or delays slow convergence if updates aren't propagated probabilistically.



multi-step, probabilistic spreading

Gossip — Solutions

- **Random peer selection** to avoid bottlenecks.
- **Push-pull exchange** to accelerate convergence.
- **Periodic rounds** for steady dissemination.
- **Rumor-mongering** to limit message load.
- **Epidemic-style propagation** for scalability.

Gossip Round Algorithm

Algorithm 7 Push-Pull Gossip

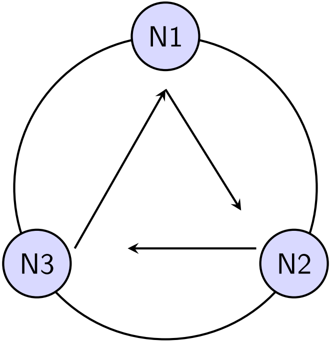
```
while true do
    wait(roundInterval)
    peer ← pickRandom()
    send(localState, peer)
    remote ← receive(peer)
    localState ← merge(localState, remote)
end while
```

Consistent Hashing

Problem

When nodes join or leave, naive sharding causes large-scale data movement. Systems need key distribution that minimizes redistribution.

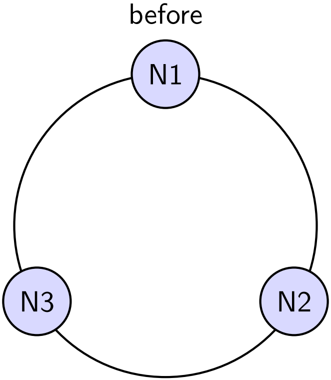
hash ring



Minimal Key Movement

Problem

When a node joins or leaves a cluster, only a small fraction of keys should be reassigned. Naive hashing remaps nearly all keys, causing disruption.



Consistent Hashing — Solutions

- **Hash ring** to place nodes on a circular space.
- **Assign keys to first clockwise node**, minimizing movement.
- **Virtual nodes** improve balancing across machines.
- **Local redistribution only** on membership change.
- **Smooth scaling** with predictable key movement.

Key Reassignment Algorithm

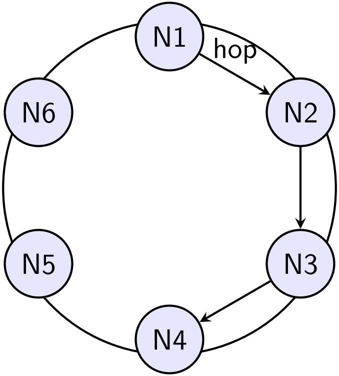
Algorithm 8 Key Redistribution on Node Join

```
onNodeJoin(newNode):
    insert(newNode, ring)
    succ ← successor(newNode)
    keys ← keysInRange(newNode, succ)
    move(keys, newNode)
```

Routing in Structured Overlays

Problem

In ring-based or tree-based overlays, nodes must route requests for keys they do not store. Routing must be efficient even with partial failures.



multi-hop key lookup

Overlay Routing — Solutions

- **Successor pointers** for basic ring traversal.
- **Finger tables** for logarithmic routing steps.
- **Route around failures** using alternate successors.
- **Lazy stabilization** to repair pointers over time.
- **Deterministic key lookup** under churn.

Finger Table Lookup Algorithm

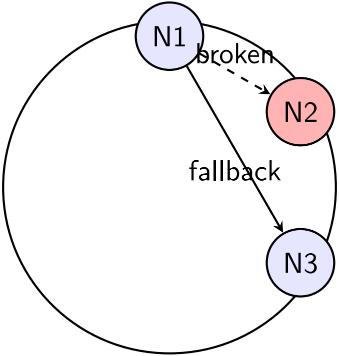
Algorithm 9 Find Successor(k)

```
findSuccessor(k):
    if k ∈ (n, successor(n)]:
        return successor(n)
    n' ← closestPrecedingFinger(k)
    forward(k, n')
```

Lookup Failures and Churn

Problem

Nodes may fail or depart unexpectedly, producing broken successor links and stale routing entries. This can cause search failures or long lookup paths.



Fault-Tolerant Lookups — Solutions

- **Multiple successors** to avoid single-point failure.
- **Periodic stabilization** to repair broken pointers.
- **Finger repair** to update stale entries gradually.
- **Redundant queries** under suspected failure.
- **Bounded lookup retries** for resilience.

Successor Stabilization Algorithm

Algorithm 10 Periodic Stabilization

```
while true do
    wait(stabilizeInterval)
    x ← successor(n).predecessor
    if x ∈ (n, successor(n)) then
        successor(n) ← x
    end if
    notify(successor(n), n)
end while
```

Summary of Key Concepts

- **Causality** via Lamport clocks, vector clocks.
- **Consistency** via anti-entropy and CRDTs.
- **Message uncertainty** from delay, loss, reordering.
- **Failure models** including crash-stop and Byzantine.
- **Scalable lookup** using gossip and consistent hashing.

Key Takeaways

- Distributed systems must tolerate **asynchrony**.
- Correctness requires **safety and liveness**.
- Causality tools structure reasoning about events.
- Replication requires robust **conflict resolution**.
- Scalable routing minimizes global coordination.

Suggested Activities

- Implement Lamport clocks for a 3-process system.
- Simulate vector clock comparisons and classify events.
- Build a mini gossip protocol and observe convergence.
- Construct a consistent hashing ring with 8 nodes.
- Analyze how lookup paths change under churn.

- A. Verma, L. Cherkasova, R. Campbell, and G. Shen, “Large-scale cluster management at google,” in *International Conference on Autonomic Computing*, 2008.
- J. Hamilton, “On designing and deploying internet-scale services,” *LISA*, 2007.
- L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- A. Verma *et al.*, “Large-scale cluster management at google with borg,” in *EuroSys*, 2015.
- S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
- B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering*. O’Reilly Media, 2016.