Cloud-Based Data Processing

Distributed Data – Part 2

Jana Giceva

Reliable cloud application

- **Identify workloads and usage requirements**
  - e.g., availability, scalability, data consistency, disaster recovery
- **Identify critical components and paths**
- **Establish availability metrics**
  - mean time to recovery (MTTR) and mean time between failures (MTBF)
  - Use these to determine when to add redundancy and to determine the SLAs to customers
- **Define the availability targets**

Availability

- **Availability = uptime = fraction of time that a service is functioning correctly**
  - “two nines” = 99% up = down 3.7 days/year
  - “three nines” = 99.9% up = down 8.8 hours/year
  - “four nines” = 99.99% up = down 53 minutes/year
  - “five nines” = 99.999% up = down 5.3 minutes/year

- **Service-Level Objective (SLO):**
  percentage of requests that need to return a correct response time within a specified timeout, as measured by the client over a certain period of time.

  e.g., “99.9% of requests in a day get a response in 200 ms”

- **Service-Level Agreement (SLA):**
  contract specifying some SLO, penalties for violation

Reliable cloud application II

- **Do a failure mode analysis (FMA)**

  identify the types of failures your application may experience and possible recovery strategies

- Create a **redundancy plan** based on the business needs and factors

- **Design for scalability** and use **load-balancing to distribute requests**

- Implement **resiliency strategy**

- **Manage the data**: store, back-up and replicate data
  - Choose the replication method
  - Document the failover and failback process
  - Plan for data recovery

- Efficient **monitoring** and fault-**recovery**

Fault-tolerance

Terminology

- **Failure:** system as a whole is not working
- **Fault:** some part of the system is not working
  - **Node fault –** crash (crash-stop/crash-recovery), deviating from algorithm (Byzantine)
  - **Network fault –** dropping or significantly delaying messages
- **Fault tolerance:**
  System as a whole continues working, despite faults.
  (some maximum number of faults assumed)
- **Single point of failure (SPOF):**
  node/network link whose fault leads to a failure

Failure detectors

- **Failure detector:**
  Algorithm that detects whether another node is faulty

- **Perfect failure detector**:
  labels a node as faulty if and only if it has crashed

- **Typical implementation for crash-stop/crash-recovery**:
  send message, await response, label node as crashed if no reply within some timeout

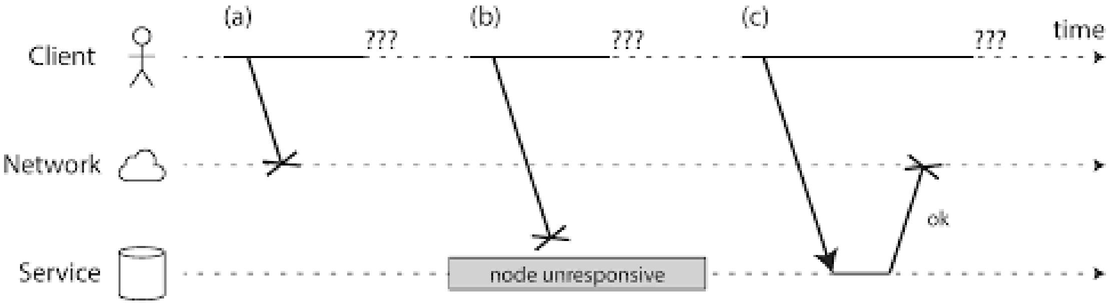
- **Problem**: cannot tell the different between
  - a crashed node,
  - temporarily unresponsive node,
  - lost message and
  - delayed message

A reliable system from unreliable components

- **No shared memory**, but **message passing** over an **unreliable network with variable delays**
- System may suffer from **partial failures**
- Each process may experience **unreliable processing pauses**
- Machines have **unreliable clocks**
- The **truth** is defined by the **majority →** requires reaching a **quorum.**

Unreliable networks and Models of distributed systems

Unreliable components (network)



- **Datacenters internal networks are asynchronous**:
  - Your request may be lost
  - Your request may be waiting in a queue and will be delivered later
  - The remote node may have failed
  - The remote node may have temporarily stopped responding, but will start responding again later
  - The remote node may have processed your request, but the response has been lost
  - The remote node may have processed your request, but the response has been delayed

- Typical we handle these problems by **sending a response message**, but even that may be lost

- **Supported with a timeout**: when to give up on waiting and assume the response is not going to arrive.

Detecting faults

- **Need to automatically detect faulty nodes:**
  - A load balancer needs to stop sending requests to a node that is dead
  - A distributed database with a single-leader replication, if the leader fails,
    one of the followers needs to be promoted to be a leader

- **Timeouts and unbounded delays**
  - How long should a timeout be?
    e.g., a short timeout detects faults faster, but can declare a node dead prematurely and cause a domino
  - Challenge: asynchronous networks (with unbounded delivery delays) and
    lack of guarantee that each server can handle requests within some maximum time.

- **Network congestion and queuing**
  - The variability of packet delays is most often due to queueing
  - Especially visible when the system is close to its maximum capacity

System models

When designing a distributed algorithm, we use a **system model** to specify our assumptions about what faults may occur.

- **Capture assumptions in a system model consisting of:**
  - **Network** behavior (e.g., message loss)
  - **Node** behavior (e.g., crashes)
  - **Timing** behavior (e.g., latency).
- **There is a specific choice of models for each of these parts.**

System model: network behavior

- **No network is perfectly reliable**
  - e.g., accidentally unplug the wrong cable, sharks and cows can cause damage and interruption to long-distance networks, or a network may be temporarily overloaded (e.g., by a DoS attack).

- Assume a bi-directional **point-to-point** communication between two nodes, with one of:
  - **Reliable** (perfect) links

    a message is received if and only if it is sent. Messages may be reordered.
  - **Fair-loss** links:

    a message may be lost, duplicated or reordered. By retrying, a message eventually gets through.
  - **Arbitrary** links (active adversary):

    a malicious adversary may interfere with messages (spy, modify, drop, spoof, replay).

- **Network partition** some links dropping / delaying all messages for an extended period of time.

System model: node behavior

Each node executes a specified algorithm, assuming one of the following:

- **Crash-stop** (fail-stop):
  a node is faulty if it crashes (at any moment). After crashing, it stops executing forever.

- **Crash-recovery** (fail-recovery):
  a node may crash at any moment, losing its in-memory state. It may resume executing, sometime later.

- **Byzantine** (fail-arbitrary):
  a node is faulty if it deviates from the algorithm. Faulty nodes may do anything, including crashing or malicious behavior.

A node that is not faulty, is called **correct.**

System model: synchrony (timing) assumptions

Assume one of the following for the network and nodes:

- **Synchronous:**
  message latency no greater than a known upper bound.
  Nodes execute algorithm at a known speed.

- **Partially synchronous:**
  The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise.

- **Asynchronous:**
  Messages may be delayed arbitrarily. Nodes can pause execution arbitrarily. No timing guarantees at all.

Violations of synchrony in practice

- **Networks usually have quite predictable latency, which can occasionally increase:**
  - Message loss requiring retry
  - Congestion/contention causing queuing
  - Network/route reconfiguration

- **Nodes usually execute code at a predictable speed, with occasional pauses:**
  - OS scheduling issues (e.g., priority inversion)
  - Stop-the-world garbage collection pauses
  - Page faults, swap, thrashing

- **Real time operating systems (RTOS) provide scheduling guarantees, but most distributed systems do not use RTOS.**

System models summary

For each of the three parts, pick one:

- **Network:**
  reliable, fair-loss, or arbitrary
- **Nodes:**
  crash-stop, crash-recovery, or Byzantine
- **Timing:**
  synchronous, partially-synchronous, or asynchronous

This is the basis for any distributed algorithm. If your assumptions are wrong, all bets are off!

Unreliability of clocks

Clocks and time in distributed systems

- **Distributed systems often need to measure time, e.g.:**
  - Schedulers, timeouts, failure detectors, retry timers,
  - Performance measurements, statistics, profiling
  - Log files and databases: record when an event occurred
  - Data with time-limited validity (e.g., cache entries)
  - Determine order of events across several nodes

- **We distinguish two types of clocks:**
  - **Physical clocks:** count number of seconds elapsed
  - **Logical clocks:** count events, e.g., messages sent

Physical clocks

- **Quartz clocks** (wristwatch, computer and phones, etc.) are cheap but not totally accurate.

- Quartz clock error: **drift**
  - One clock runs slightly faster, another slower
  - Drift is measured in parts per million (ppm).

    1 ppm = 1 microsecond/second = 86 ms/day = 32s/year
  - Most computer clocks correct within 50 ppm

- For greater accuracy, use **atomic clocks**.
- **Leap seconds** – to keep the UTC and TAI in sync (linked to the rotation of earth)

- **Computers and time**
  - Unix time: number of seconds since 1 January 1970 (epoch) – not counting leap seconds
  - ISO 8601: year, month, day, hour, minute, second and timezone offset relative to UTC

- **To be correct, software that works with timestamps needs to know about leap seconds.**

Clock synchronization

- Computers track physical time/UTC with a quartz clock

- Due to **clock drift**, clock error gradually increases.

- **Clock skew:** difference between two clocks at a point in time

- **Solution:** periodically get the current time from a server that has a more accurate time source (atomic clock or GPS receiver)

- **Protocols: Network Time Protocol (NTP), Precision Time Protocol (PTP)**
  - Make multiple requests to the same server,
    use statistics to reduce error due to variations in network latency
  - Reduces clock skew to a few milliseconds in good network conditions.

Time-of-day and monotonic clocks

```
// BAD
long startTime = System.currentTimeMillis();
doSomething();
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

← NTP client steps the clock during this

```
// GOOD
long startTime = System.nanoTime();
doSomething();
long endTime = System.nanoTime();
long elapsedNanos = endTime - startTime;
// elapsedNanos is always >= 0
```

Time-of-day and monotonic clocks

- **Time-of-day clock:**
  - Time since a fixed date (e.g., 1 January 1970 epoch)
  - **May suddenly move forwards or backwards** (NTP stepping), **subject to leap second adjustments**
  - **Timestamps can be compared across nodes (if synced)**
  - Java: System.curretTimeMillis()
  - Linux: clock_gettime(CLOCK_REALTIME)

- **Monotonic clock:**
  - Time since arbitrary point (e.g., when the machine booted up)
  - **Always moves forward at near constant speed**
  - **Good for measuring elapsed time on a single node**
  - Java: System.nanoTime():
  - Linux: clock_gettime(CLOCK_MONOTONIC)

Clock readings should have a confidence interval

- **When getting the time from a server, the uncertainty is based on:**
  - the expected quartz drift since your last sync,
  - the reference server’s uncertainty,
  - and the network round-trip time to the reference server.

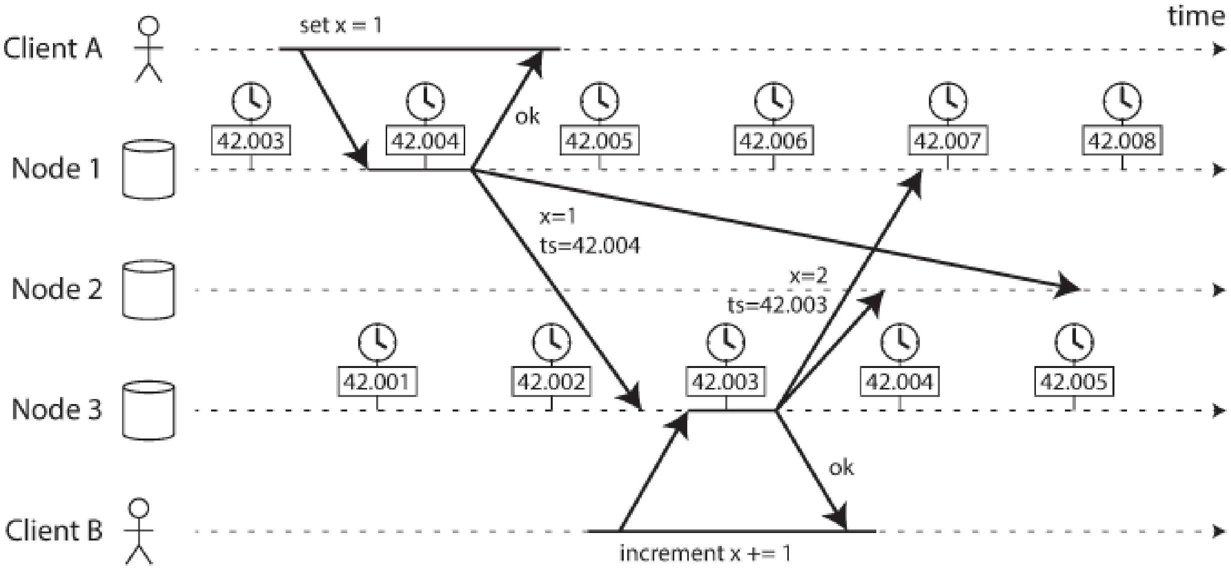
  e.g., A system may be 90% confident that the time now is between 10.3 and 10.5 seconds past the minute.

- **Most systems do not expose this uncertainty**

  Notable exception: Google’s TrueTime API, which explicitly reports the confidence interval on the local clock.
  - When you ask it for the current time, you get back

    two values [earliest, latest], which are the earliest possible and the latest possible timestamp.
  - Used in Spanner (to be covered in a few weeks).

Ordering of messages



Logical vs. physical clocks

- **Physical** clock: count number of **seconds elapsed**
- **Logical** clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality.**

Logical clocks: designed to **capture causal dependencies**

$$(e_1 \rightarrow e_2) \xrightarrow{\text{yields}} (T(e_1) < T(e_2))$$

Distributed systems/algorithms typically cover two types of logical clocks:

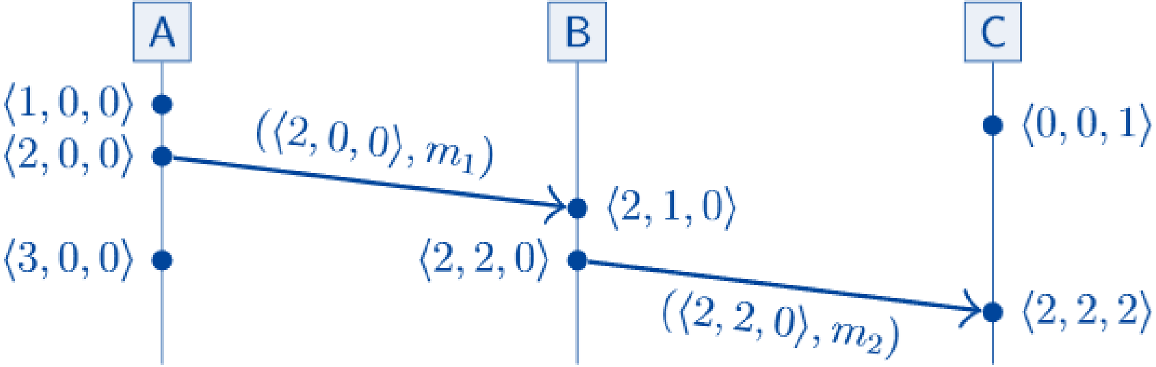
- **Lamport** clocks
- **Vector** clocks

Vector clocks

- **When we want to detect concurrent events, we use vector clocks:**
  - Assume n nodes in the system, $N \ = < N_1, N_2, \dots, N_n >$
  - Vector timestamp of event a is $V(a) \ =< t_1, t_2, \dots, t_n >$
  - t_i, is number of events observed by node N_i
  - Each node has a current vector timestamp T
  - On event at node N_i, increment vector element $T[i]$
  - Attach current vector timestamp to each message
  - Recipient merges message vector into its logical vector

Vector clocks example

- Assuming the vector of nodes is



- The vector timestamp of an event e represents a set of events: e and its causal dependencies: $\{e\} \cup \{a \mid a \rightarrow e\}$

- For example, $\langle 2,2,0\rangle$ represents the first two events from A, the first two events from B,and no events from C

Majority decides the **truth**

- In a distributed system, the **truth** is defined by the majority
  - A single node cannot trust its own judgement of a situation
  - Many distributed algorithms rely on a **quorum,** i.e., voting among the nodes.
    - Including when to declare a node as dead
  - Quorums are especially important for our upcoming discussion on consensus (next week).

Broadcast protocols

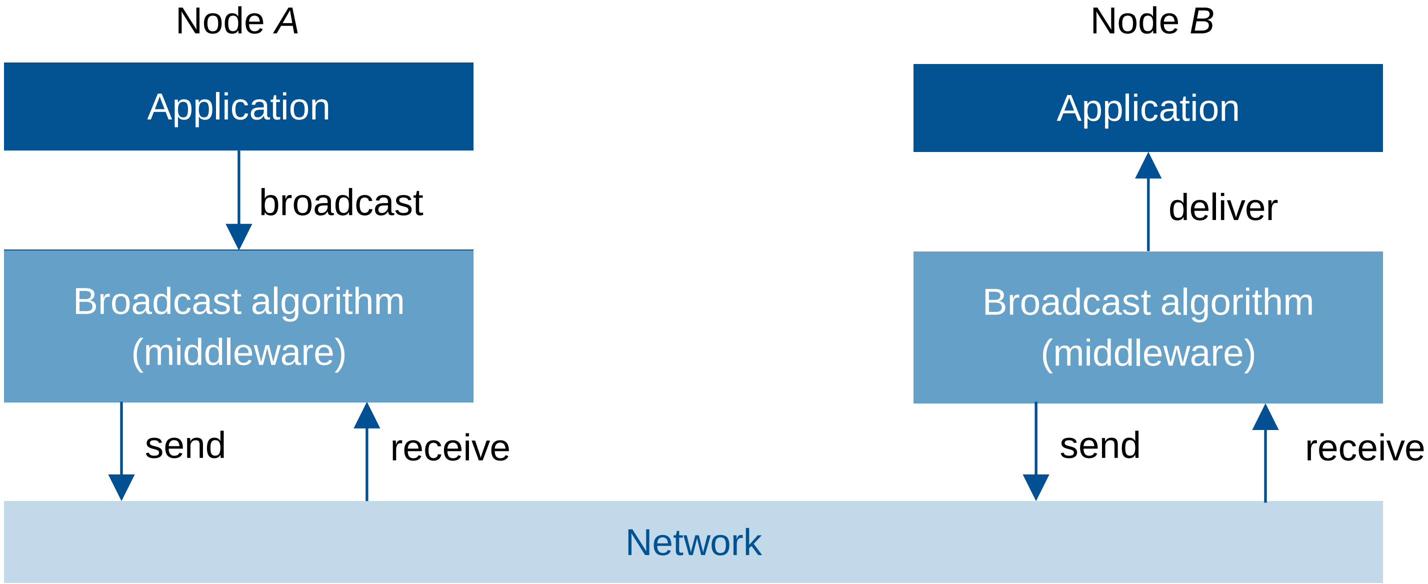
Broadcast protocols

- Broadcast (multicast) is a **group communication**:
  - One node sends message, all nodes in the group deliver it
  - Set of group members may be fixed (static) or dynamic
  - If one node is faulty, remaining group members carry on

- Build upon system models:
  - Can be **best-effort** (may drop messages) or
    reliable (non-faulty nodes deliver every message by retransmitting dropped messages).

  - Asynchronous/partially synchronous timing model
    → **no upper bound** on message latency

Receiving versus delivering



- Assume network provides point-to-point send/receive.
- After broadcast algorithm receives a message from the network, it **may buffer/queue** it **before delivering** to the application.

Forms of reliable broadcast

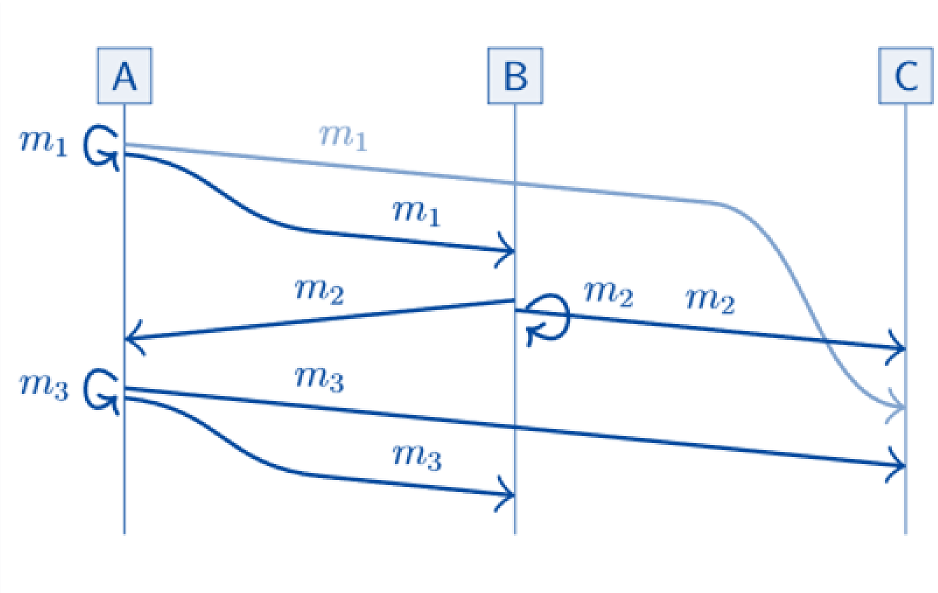
- **FIFO broadcast**
  if m_1 and m_2 are broadcast by the application of the same node, and broadcast(m_1) $\rightarrow$ broadcast (m_2), then m_1 must be delivered before m_2

- **Causal broadcast**
  if broadcast(m_1) $\rightarrow$ broadcast (m_2), then m_1 must be delivered before m_2

- **Total order broadcast**
  if m_1 is delivered before m_2 on one node, then m_1 must be delivered before m_2 on all nodes

- **FIFO-total order broadcast**
  combination of FIFO broadcast and total order broadcast

FIFO broadcast

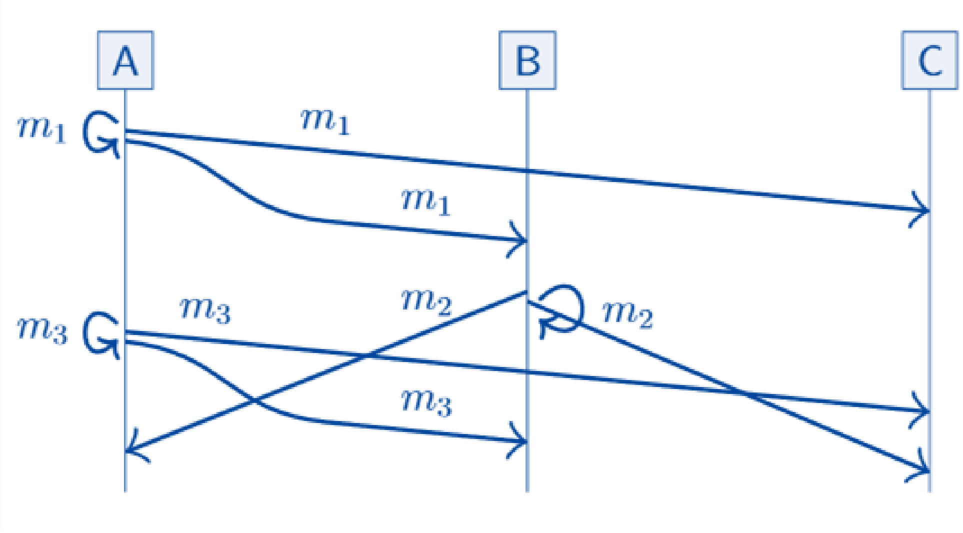


- **Messages sent by the same node must be delivered in the order they were sent.**

- Messages sent by different nodes can be delivered in any order.

- Valid orders:
  (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

Causal broadcast

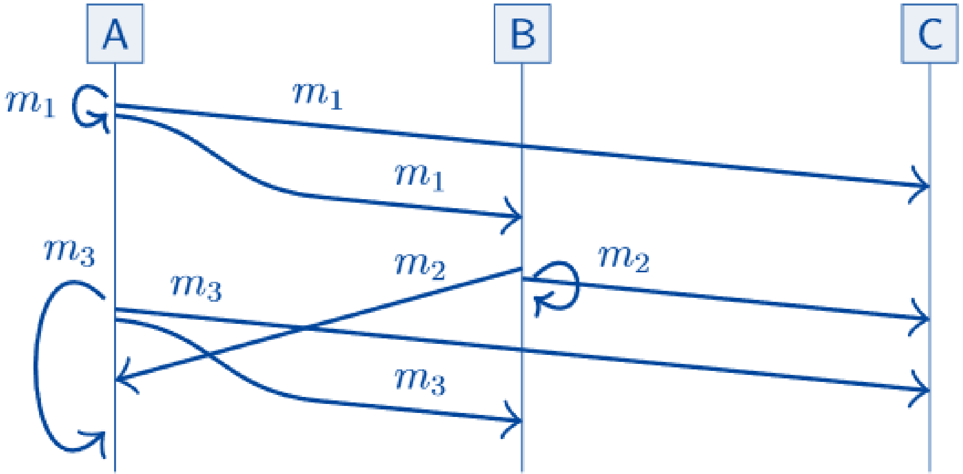


- **Causally related messages must be delivered in causal order.**

- Concurrent messages can be delivered in any order.

- Here:
  broadcast(m_1) → broadcast (m_2) and
  broadcast(m_1) → broadcast (m_3)
  →
  valid orders are
  (m_1, m_2, m_3) or (m_1, m_3, m_2)

Total order broadcast



- **All nodes must deliver messages in the same order**
  here (m_1, m_2, m_3)
- This includes a node's delivery to itself.

Total order broadcast algorithms

- **Single leader** approach:
  - One node is designated as a leader
  - To broadcast message, send it to the leader: leader broadcasts it via FIFO broadcast
  - Problem: leader crashes → no more messages delivered
  - Changing the leader safely is difficult

- **Logical clocks** approach:
  - Attach a vector timestamp to every message
  - Deliver messages in total order of timestamps
  - Problem: how do you know if you have seen all messages with timestamp <T?
    - Need to use FIFO links and wait for message with timestamp >=T from every node.

- In both approaches a crash from a single node can stop all other nodes from being able to deliver messages.
- Need a fault-tolerant total order broadcast.

Replication using broadcast

- Last week’s replication was “implemented” using the **best-effort broadcast**:
  a client broadcasts every read or write to all of the replicas,
  but the protocol is **unreliable** (requests may be lost) and provides **no ordering guarantees.**

- **Replication with total order broadcast:**
  every node delivers the **same messages** in the **same order**

- **State machine replication (SMR):**
  - **FIFO-total order broadcast** every update to all replicas
  - Replica delivers update message: apply it to own state
  - Applying an update is deterministic
  - Replica is a **state machine:**
    starts in a fixed initial state, and
    goes through same sequence of state transitions in the same order
    → all replicas end up in the same state

State machine replication

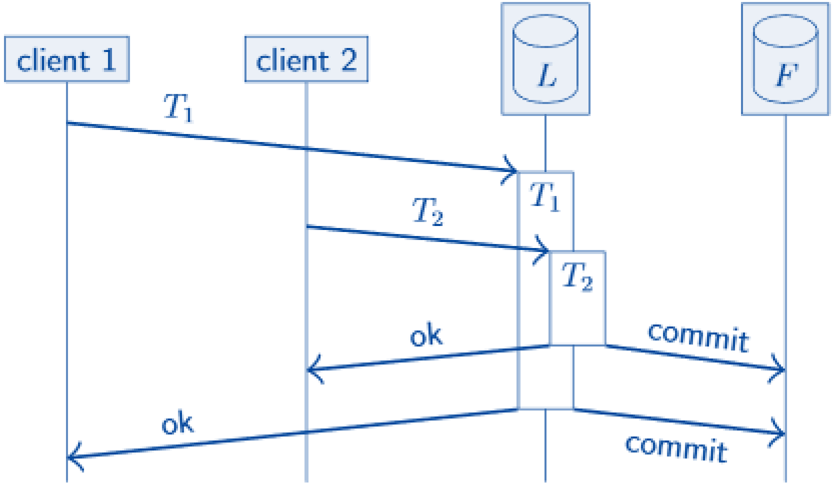
```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic
end on
```

- **Closely related ideas:**
  - Serializable transactions (execute in delivery order)
  - Blockchains, distributed ledgers, smart contracts

- **Limitations:**
  - Cannot update state immediately, have to wait for delivery through broadcast
  - Need fault-tolerant total order broadcast (next week)!

Database leader replication

- Leader database replica, ensures total order broadcast.
- Follower F applies the transaction log in commit order.



Replication using causal (and weaker) broadcast

- State machine replication uses (FIFO-) total order broadcast.
- Can we use weaker forms of broadcast too?

- If replica state updates are **commutative,** replicas can process updates in different orders and still end up in the same state.

- Updates f and g are commutative if $f\big(g(x)\big) = g\big(f(x)\big)$

broadcast	assumptions about state update function
Total order	Deterministic (SMR)
Causal	Deterministic, concurrent updates commute
Reliable	Deterministic, all updates commute
Best-effort	Deterministic, commutative, idempotent, tolerates message loss

References

The material covered in this class is mainly based on:

- The book *“Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”* by Martin Kleppmann (Chapters 8 and part of 9) (link)
- Slides from “*Distributed Systems”* course from University of Cambridge (link)

Some information about application-level design were based on material from:

- Microsoft’s Azure Application Architecture Guide
  - Design Reliable Applications (link)
  - Design for self-healing (link)