

Exercise walk-through

9.7

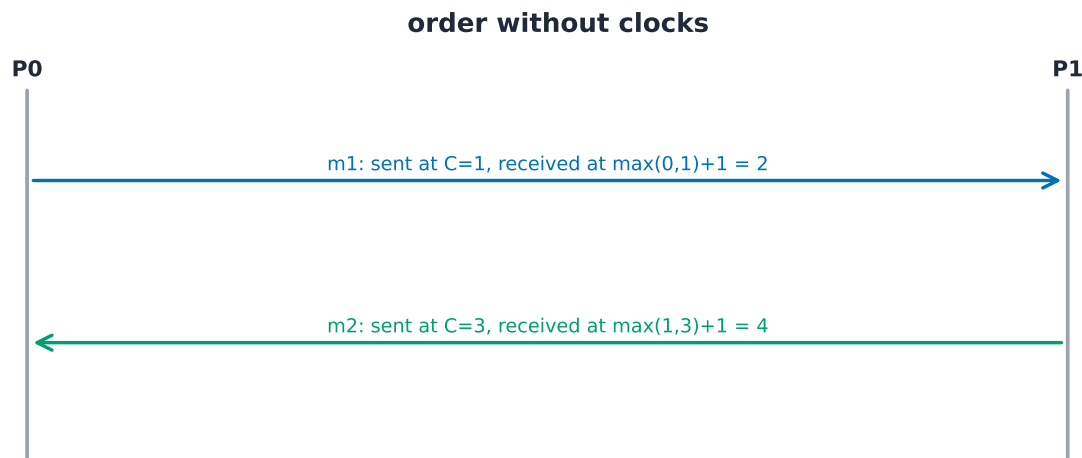
Distributed systems fundamentals

University Computer Science

Distributed: order without clocks

- 1 **Clocks** P0 sends at C 1; P1 receives at $\max(0,1)+1 = 2$; P1 sends at 3; P0 receives at $\max(1,3)+1 = 4$ - causality preserved, time not used.
- 2 **One-way** If a **happens-before** b then $C(a) < C(b)$; the converse is false - equal clocks imply nothing.
- 3 **Retries** deduct(user, 25, id=xyz) twice deducts once: the ledger records xyz as done - idempotent by key, not by hope.

Message ordering diagrams



causality only: if a happens-before b then $C(a) < C(b)$ - the converse is false

idempotent operations plus a dedupe ledger make retries safe; a timeout means unknown, not failed

Lamport counters across two processes: 1, 2, 3, 4 with causality intact.

Check yourself

- Headers around 100 payload bytes (Ethernet+IP+**TCP**)...? (54.)
- IHL 5 means a header of...? (20 bytes.)
- Course graph: F settles at...? (13.)
- P1 sends at C 3 to P0 at C 1: P0 becomes...? (4.)

Exercises 2.1-2.3

2.1 P1 at $C = 5$ sends to P0 at $C = 2$. What is P0's next C ? [3]

2.2 Can two events share a Lamport value? [3]

2.3 What does idempotence guarantee for a retried operation? [3]

Exercises 2.1-2.3

2.1 P1 at $C = 5$ sends to P0 at $C = 2$. What is P0's next C ? [3]

2.1

- $C = \max(2, 5) + 1 = 6$ ✓.

2.2 Can two events share a Lamport value? [3]

2.2

- Yes – at different processes; equal clocks imply nothing about order ✓.

2.3 What does idempotence guarantee for a retried operation? [3]

2.3

- Executing it more than once has the same effect as once ✓.

Exercise 3.1

3.1 P0 ($C = 3$) and P1 ($C = 7$) exchange: P0 sends m_1 ; P1 receives and immediately sends m_2 ; P0 receives m_2 ; P0 then sends m_3 .

Compute the Lamport value attached to each message and each process's counter after every event (list the sequence). State which pairs are ordered by happens-before and which are concurrent. State why wall clocks could not have produced a guaranteed order. [8]

Exercise 3.1

3.1 P0 ($C = 3$) and P1 ($C = 7$) exchange: P0 sends m_1 ; P1 receives and immediately sends m_2 ; P0 receives m_2 ; P0 then sends m_3 .

Compute the Lamport value attached to each message and each process's counter after every event (list the sequence). State which pairs are ordered by happens-before and which are concurrent. State why wall clocks could not have produced a guaranteed order. [8]

3.1

- m_1 sent at $C = 4$ (P0: $3 \rightarrow 4$) ✓.
- P1 receives: $C = \max(7, 4) + 1 = 8$ ✓.
- m_2 sent at $C = 9$ (P1: $8 \rightarrow 9$) ✓.
- P0 receives: $C = \max(4, 9) + 1 = 10$; m_3 sent at 11 ✓.
- Ordered: $\text{send}(m_1)$ before $\text{receive}(m_1)$, before $\text{send}(m_2)$, before $\text{receive}(m_2)$, before $\text{send}(m_3)$ – a causal chain ✓.
- Concurrent: none in this trace – every event lies on one chain; concurrent events would share no arrow path ✓.
- Wall clocks skew without bound across machines: a later event can carry an earlier timestamp, so no guaranteed order ✓.

Exercise 3.2

3.2 A payment service exposes `deduct(user, amount, id)` and clients retry on timeout.

State the double-charge risk and when it fires. Give the idempotent design (key plus ledger) and what the retry returns. State what a timeout proves and what the client should do. Give the rule for choosing which operations must be idempotent. [7]

Exercise 3.2

3.2 A payment service exposes `deduct(user, amount, id)` and clients retry on timeout.

State the double-charge risk and when it fires. Give the idempotent design (key plus ledger) and what the retry returns. State what a timeout proves and what the client should do. Give the rule for choosing which operations must be idempotent. [7]

3.2

- Risk: the request lands, the reply is lost, the client retries – two deductions fire ✓.
- Design: idempotency key `id` plus a ledger – the first execution records `id` as done; the retry looks it up and returns the original result without re-executing ✓.
- A timeout proves nothing about execution – the client retries, and safely, because the operation is idempotent ✓.
- Rule: every operation a client may retry (any network call) must be idempotent ✓.