Ciências
ULisboa

Programação em Sistemas Distribuídos

MEI-MI-MSI

2018/19

2. Distributed Systems Paradigms

Prof. António Casimiro

Review of basic
Distributed systems paradigms

Naming and addressing

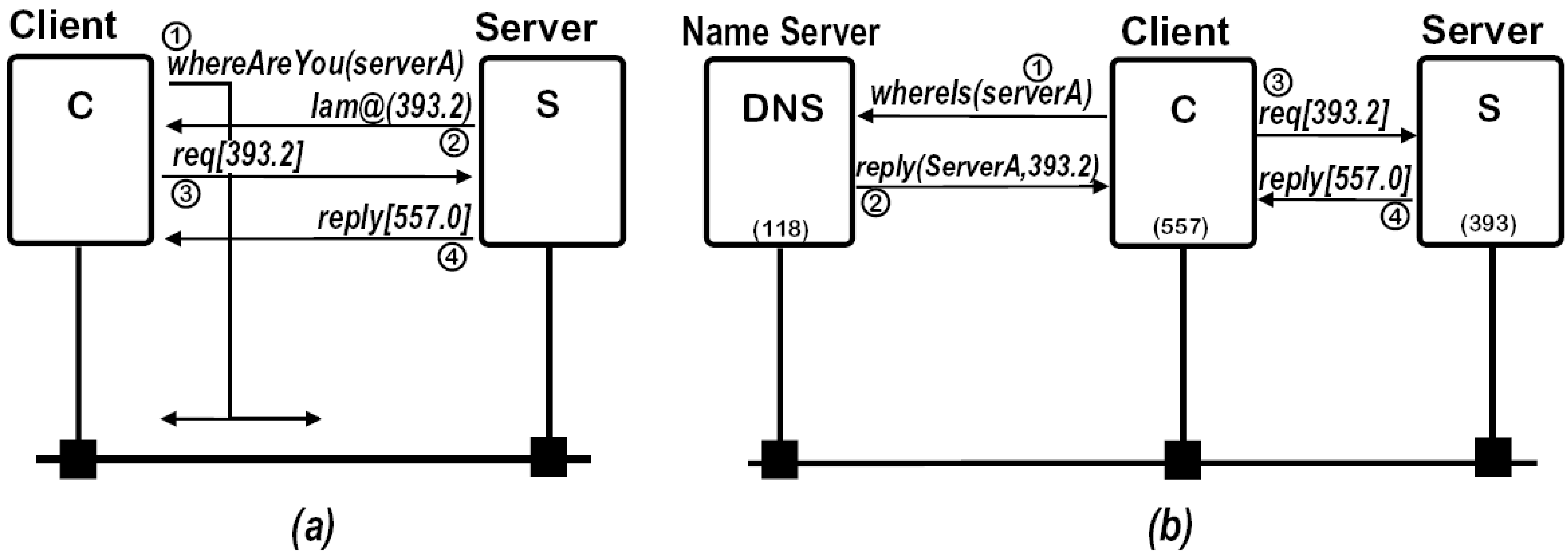


Figure 2.2. Name to Address Translation: (a) Broadcast; (b) Name Server

Message passing

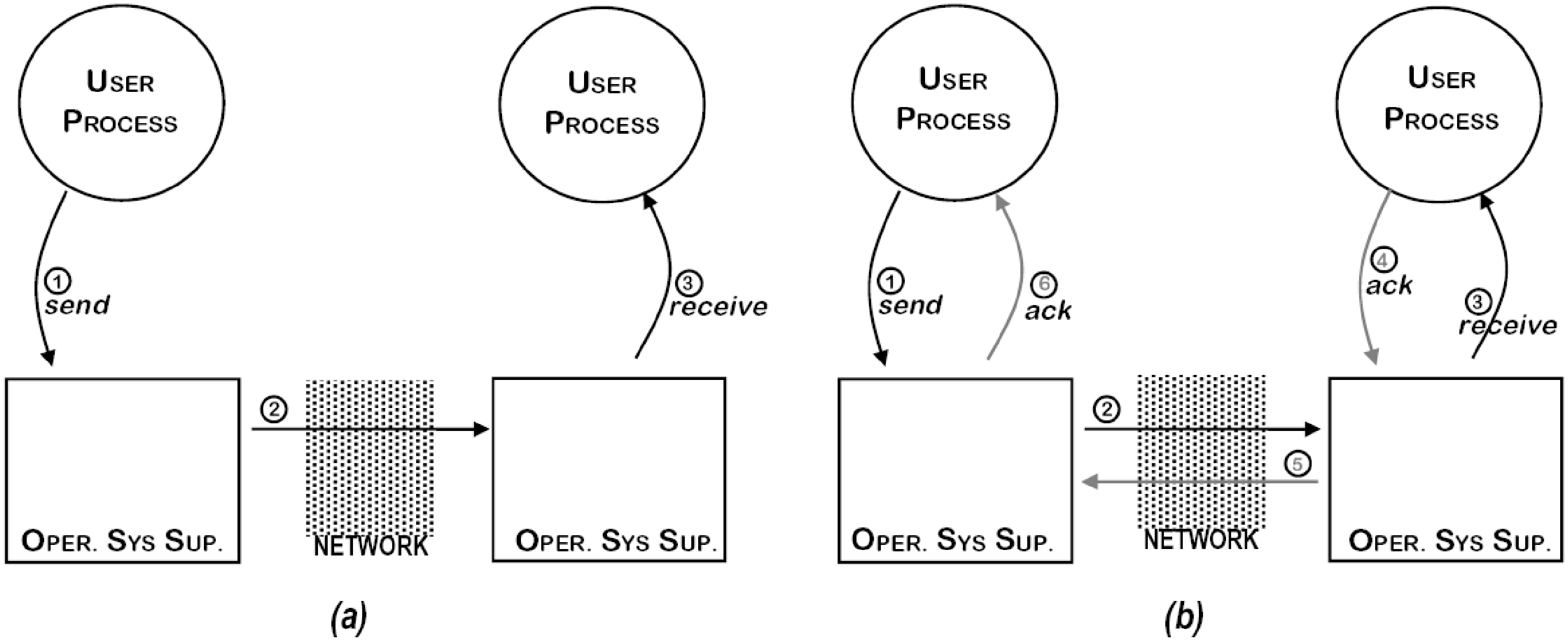


Figure 2.4. Message Passing Protocols: (a) Send-Receive;(b) Acknowledged-Send

Message passing

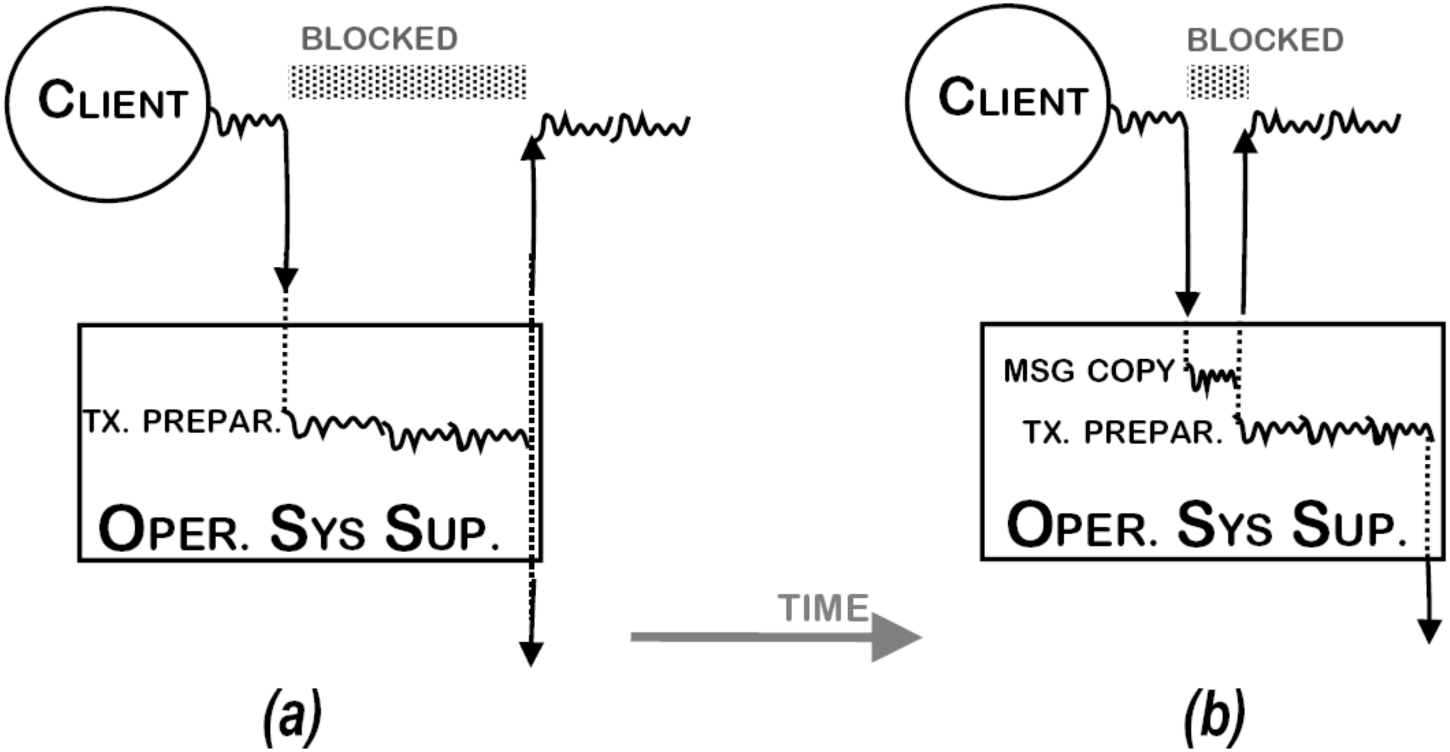


Figure 2.5. Message Send Blocking: (a) User Buffer; (b) Driver Buffer

Remote operations

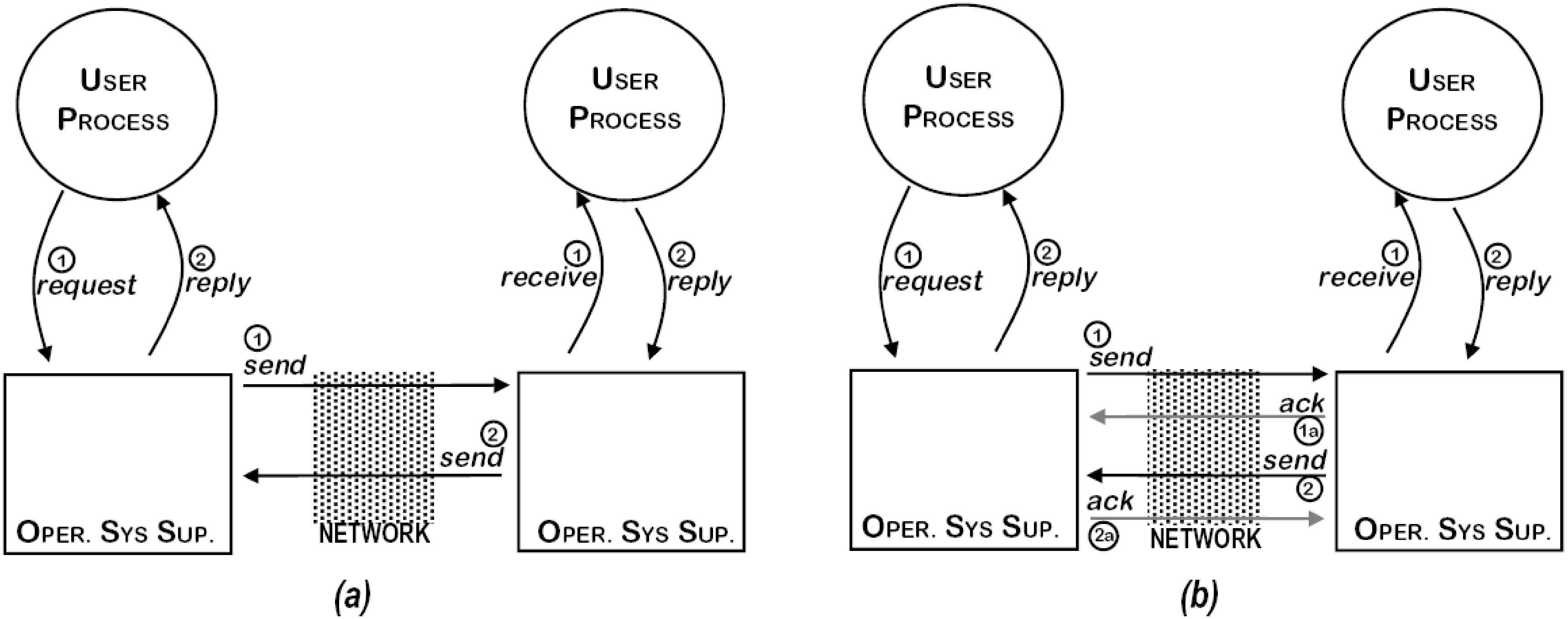


Figure 2.6. Remote Operation Protocols: (a) Plain Request-Reply; (b) Acknowledged

Remote operations

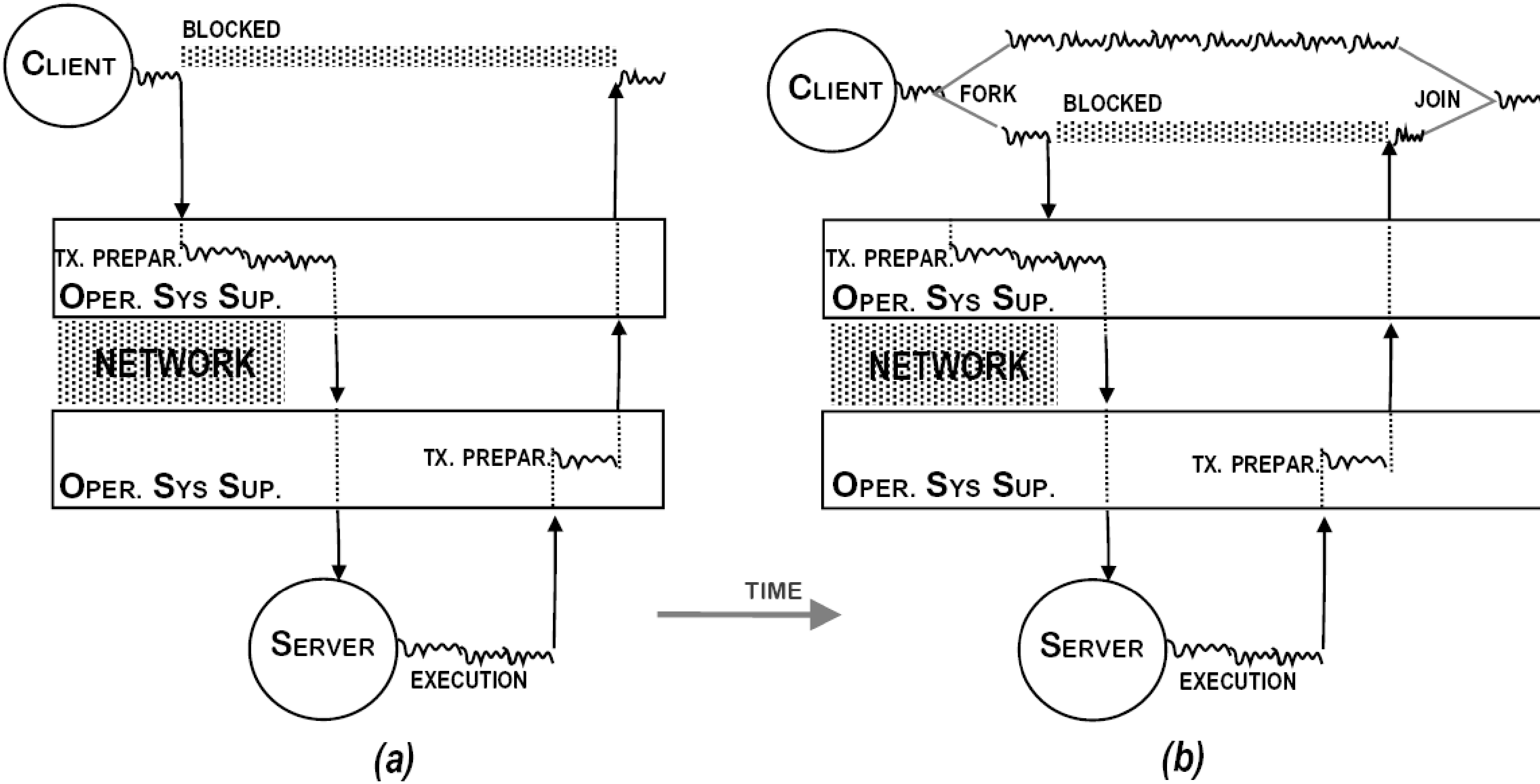


Figure 2.8. Remote Operation Interfaces: (a) Blocking; (b) Non-Blocking

Multicast

- Process groups
- Group communication service
- Group membership (views)
- Main components of a multicast protocol:
  - routing
  - omission tolerance
  - flow-control
  - ordering
  - failure recovery

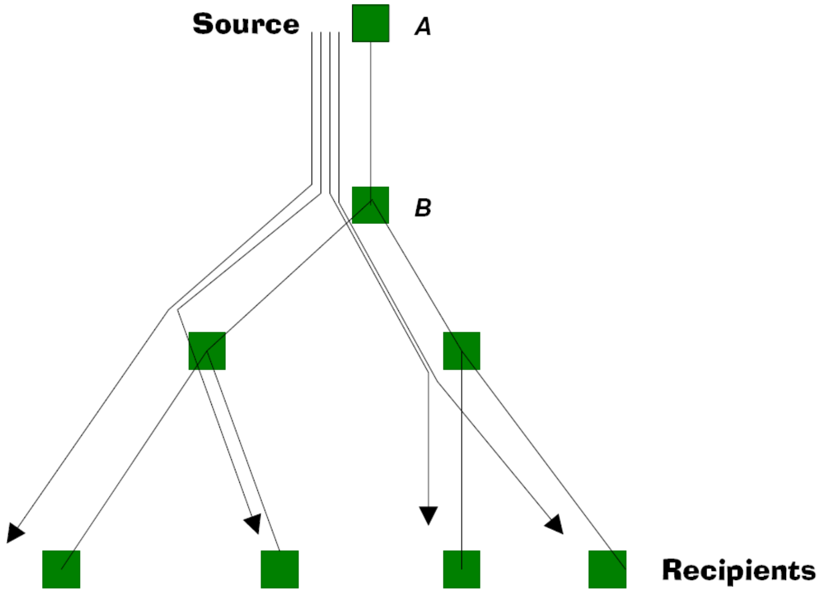


Figure 2.9. Multicast Tree

Advanced
Distributed systems paradigms

Time and Clocks

Time and Clocks

- Common uses of clocks in distributed systems:
  - Trigger events
  - Register the time at which events occurred
  - Measure durations
- Examples:
  - File timestamping
  - Measuring benchmark program speed
- Artifact: support protocol implementation
  - Timers and clocks
  - Message ordering

Global Time
Why?

- **Trigger events**
  - How do you synchronise distributed event triggering?
- **Register the time at which events occurred**
  - How do you correlate distributed registers?
- **Measure durations**
  - How do you measure what starts here and ends there?

- **Global Clock**:
  - Abstraction: a set of mutually synchronised local clocks

Absolute Time
Why?

- **Coordination of systems that do not communicate directly**
- **Bounding the error in lengthy duration measurement**

- **Absolute global clock**:
  - Abstraction: a set of local clocks synchronised individually to a common reference, which besides should be universal
  - E.g., UTC- Universal Time Coordinated; TAI- Temps Atomique Internacional

Time and clocks
Properties of a Global Clock System

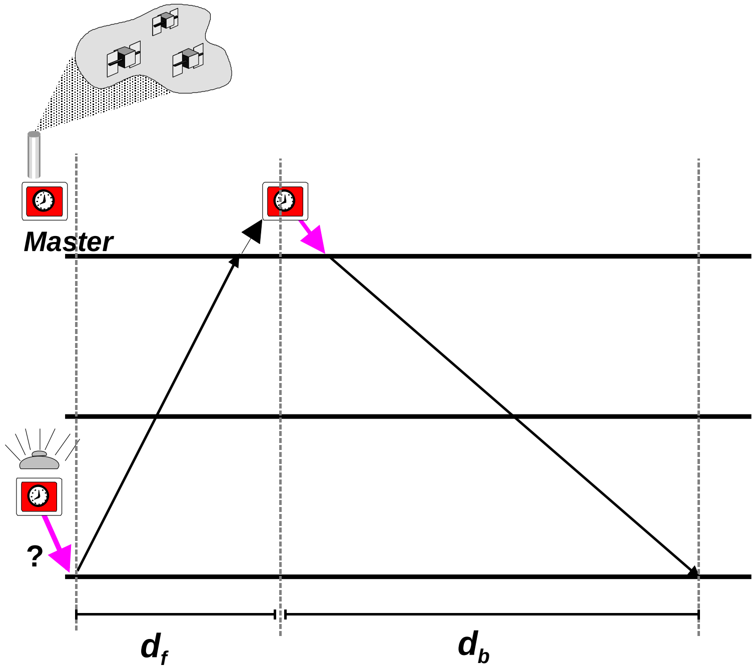
- Physical Granularity (g)
  - fundamental tick or pulse of hardware clock
- Virtual Granularity (gv)
  - tick of the virtual clock, multiple of g
- Convergence (δv)
  - measures how close virtual clocks are to each other immediately after the synchronization algorithm terminates
- Precision (Πv)
  - measures how closely virtual clocks remain synchronized to each other at any time
- Rate (ρv)
  - instantaneous rate of drift of virtual clocks
- Envelope Rate (ρa)
  - long-term, or average rate of drift
- Accuracy (αv)
  - measures how closely virtual clocks are synchronized to an absolute real time reference, provided externally

Clock synchronisation

- Hardware clocks drift with time
  - Some (e.g. cesium or rubidium GPS clocks) are extremely stable
  - But PC and workstation HW clocks are bad (worse than 1ppm)
- So they have to be synchronised periodically
  - **Clock synchronisation protocols**
- Internal synchronisation:
  - Ensures **precision**
  - Normally clocks cooperatively readjust (*agreement or convergence based*)
- External synchronisation:
  - Ensures **accuracy and precision** ($\pi_v = 2\ \alpha_v$)
  - normally clocks read from an external master (*round-trip-based*)

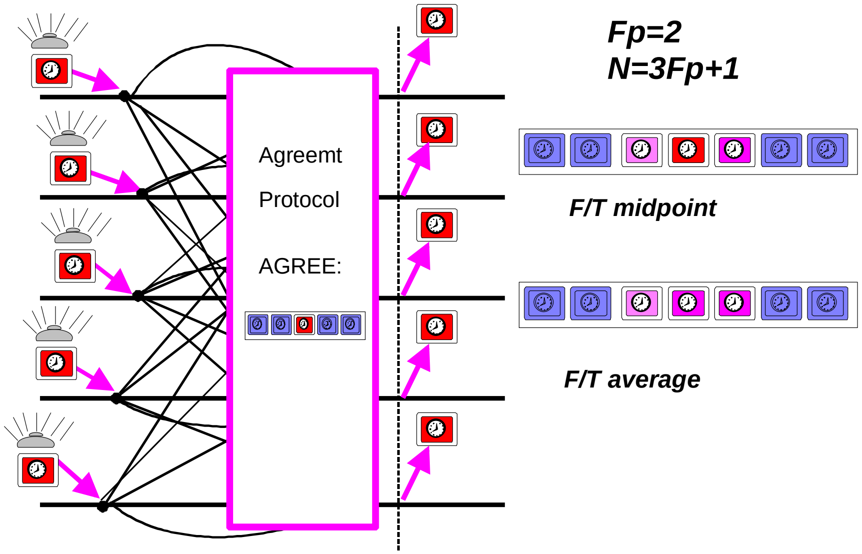
Clock synchronisation
Master- or round-trip based

- External synchronization:
  - Based on round-trip measurement from central master clock
- Accuracy:
  - Assessed by measuring round-trip delay
  - Depends on delay symmetry
  - Best run when (df+db)/2 ≈ df ≈ db
- Precision:
  - Precision is twice worse than accuracy ($\pi_v = 2\ \alpha_v$)



Clock synchronisation
Agreement-based

- Agreement based:
  - Convergence based on F/T average or median
  - Clocks are Byzantine, to represent value faults
- Precision:
  - Precision depends on **clock reading error**
  - Processors compute common value to set clocks to
  - Time for agreement is in critical path of precision

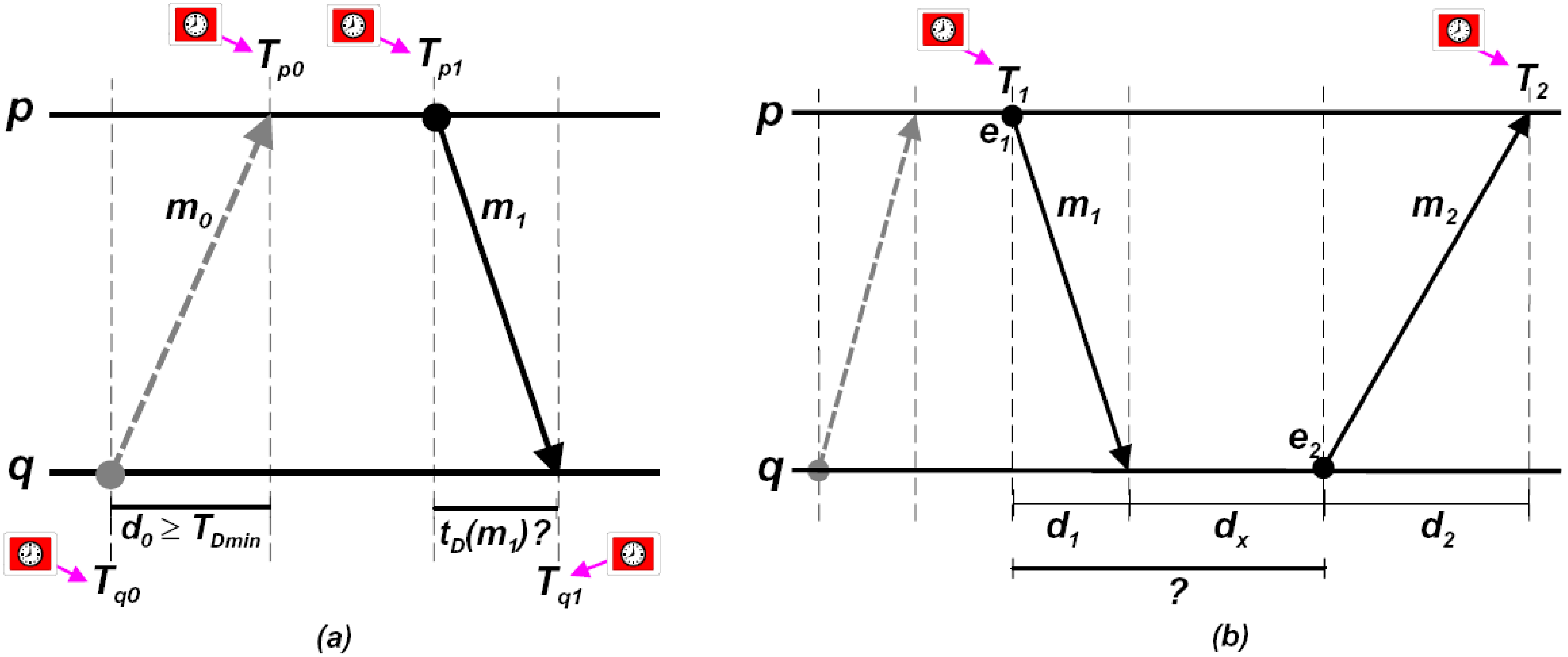


Duration measurement errors

- **Local** duration measurement between *a* e *b*, using local timestamps (ignoring ρ)
  - Tb - Ta = t(b) - t(a) $\pm \varepsilon$, $\mathbf{0 \leq \varepsilon \leq g}$
- **Distributed** duration measurement between *a* e *b*, using global timestamps (ignoring ρ)
  - Tb - Ta = t(b) - t(a) $\pm \varepsilon$, $\mathbf{0 \leq \varepsilon \leq \pi + g}$
- Roundtrip duration measurement
  - Based on a message ping-pong, avoids using explicitly synchronised clocks, at the cost of a potentially higher error

Time and clocks

Round-trip Duration Measurement

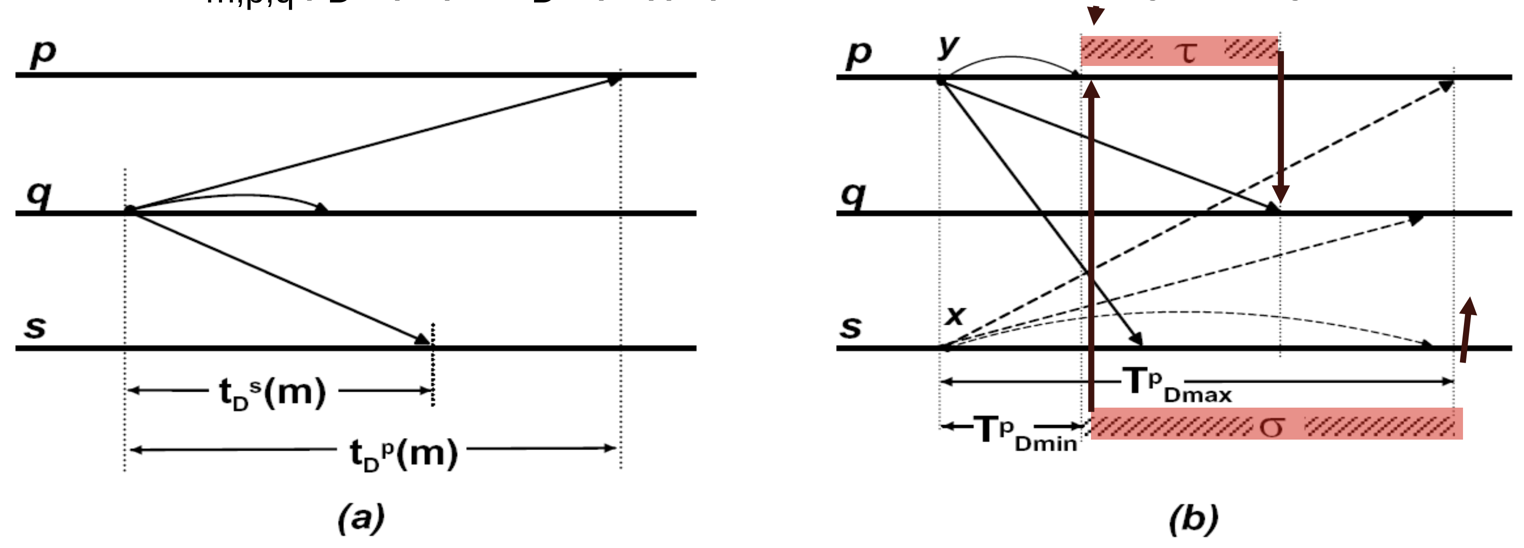


(a) Message delay ***(b) Distributed Duration***

Synchronism

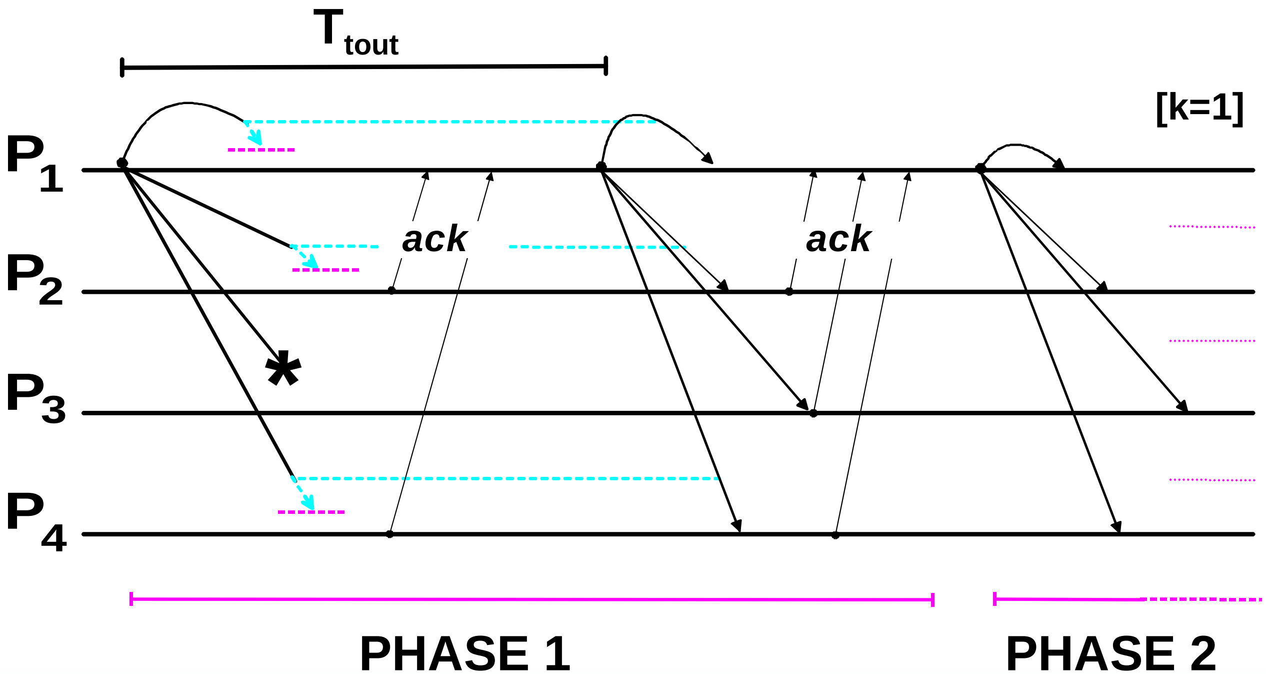
Synchronism

- Synchronism means (w.r.t. messages):
  - Known and bounded message delivery delay
- Synchronism metrics of quality:
- Steadiness (σ) *[estabilidade]*
  - $\sigma = max_p(T_{Dmax} - T_{Dmin})$ (variance of delivery delay across execs)
- Tightness (τ) *[rigidez]*
  - $\tau = max_{m,p,q}(t_D^p(m) - t_D^q(m))$ (variance of delivery delay in same exec)



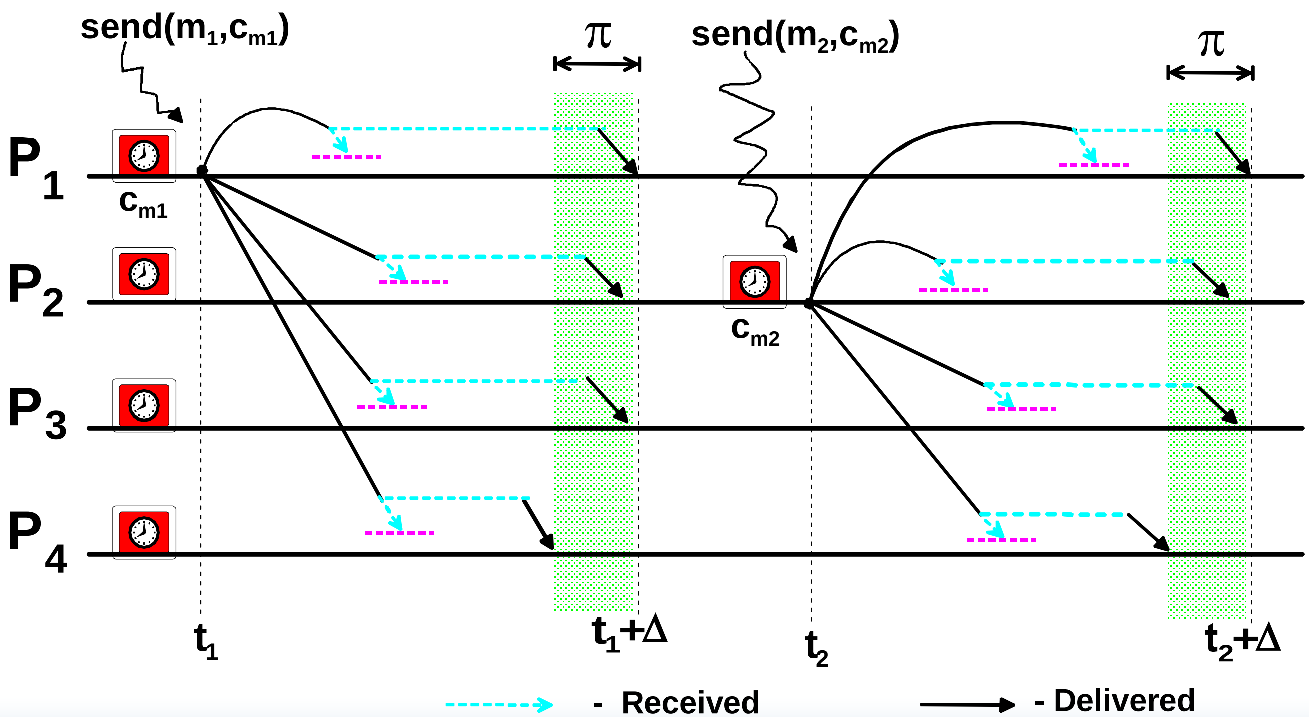
Degrees of steadiness and tightness

- **Protocol with little steadiness and tightness**
- Trade-off: tolerates omissions up to *k* (retransmits *k+1* times)
- Consider δ_{min} and δ_{max} as min and max delivery time, resp., and $T_{tout} \geq 2\delta_{max}$ the re-transmission timeout
- Then delivery delays are: $T_{Dmax} \leq (k+1)T_{tout}$ and $T_{Dmin} \geq \delta_{min}$



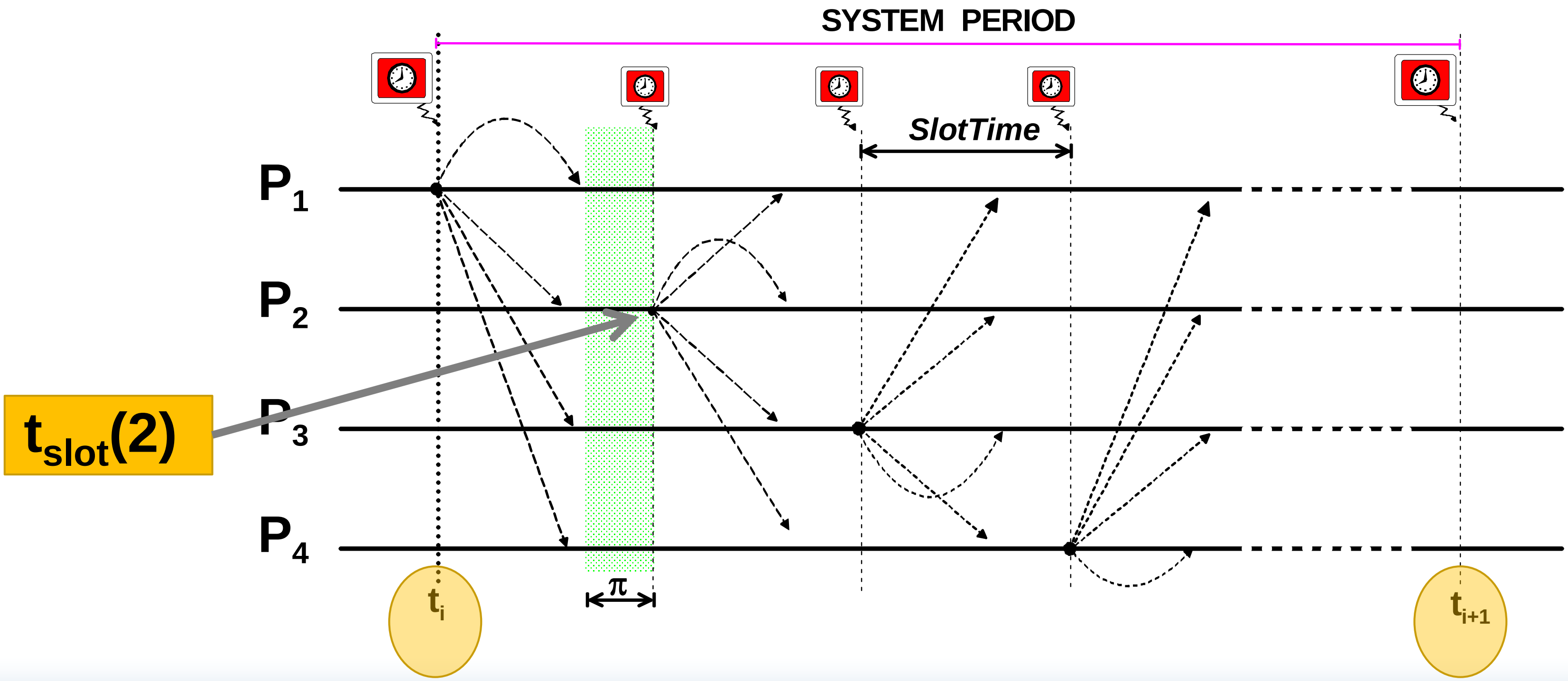
Degrees of steadiness and tightness

- **Protocol with high steadiness and tightness, of the Δ class**
- Receives message and waits $t_{send} + \Delta$ before delivering
  - Everywhere "at the same time" (tightness), with same delay Δ (steadiness)
  - Any two messages delivered in the order of sending (clock time)



Degrees of steadiness and tightness

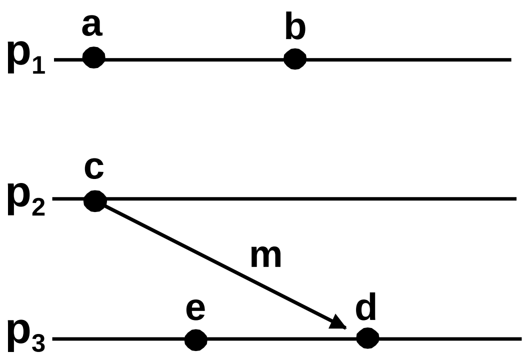
- **Protocol with high steadiness and tightness, of the TDMA class**
- P_k waits for $t_{slot}(k)$ before transmitting
  - Short broadcast medium enforces fair transmission tightness and steadiness
  - But delivery tightness and steadiness enforced by lattice periods *i , i+1, ...*
  - Any two messages delivered within a same period *i* are concurrent, message delivered in period *i* precedes any message delivered in period *i+1*

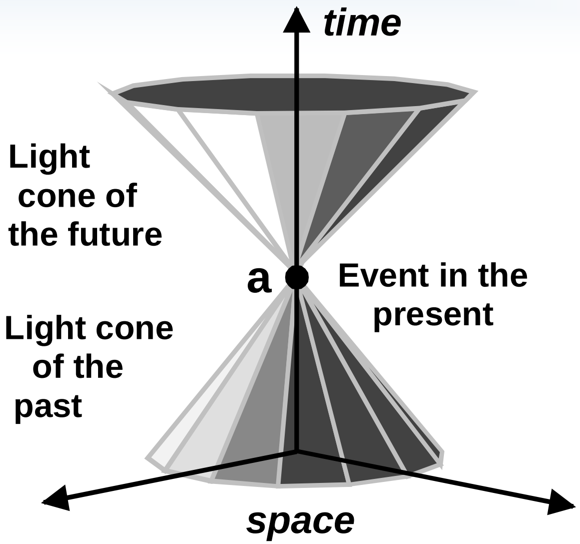


Ordering

Causal Order

So, is c → e ?





- **Happened-before relation:**
  - *a* → *b* iff:
  - *a* before *b* locally
  - *a* send and *b* reception of *m*

- **Cause-effect order:**
  - natural universe order
- **A partial order:**
  - depends of time-like and space-like separation of events
  - relativistic effect due to speed difference between local and message events

Ordering

- **Causal Delivery**
  - For any two messages M1 and M2, **sent by p and q**, delivered to any correct processes,
    if $send(M1) \rightarrow send(M2)$, then $deliver(M1) \rightarrow deliver(M2)$
  - **Example:** clients compete over a server to schedule a trip, buy some stock, and communicate between them at the same time; only causal order reflects the inter-client relations on the server requests
- **FIFO Delivery** (first-in-first-out)
  - For any two messages M1 and M2, **sent by p**, delivered to any correct processes,
    if $send(M1) \rightarrow send(M2)$, then $deliver(M1) \rightarrow deliver(M2)$
  - **Example:** this is a reduction of the general causal order to messages originated from only one sender (e.g. TCP ordering)

Causal Ordering implementations

- Logical implementations are usual (LC, VC)
  - A message m1 logically precedes ($\xrightarrow{\mathcal{L}}$) m2 iff: m1 is sent before m2 by the same participant or m1 is delivered to the sender of m2 before it sends m2 or there exists m3 s.t. m1 $\xrightarrow{\mathcal{L}}$ m3 and m3 $\xrightarrow{\mathcal{L}}$ m2
- Why does logical order work?
  - if participants only exchange information by sending and receiving messages through a given protocol, causality is developed only through those messages

Is logical ordering always faithful? **NO!**

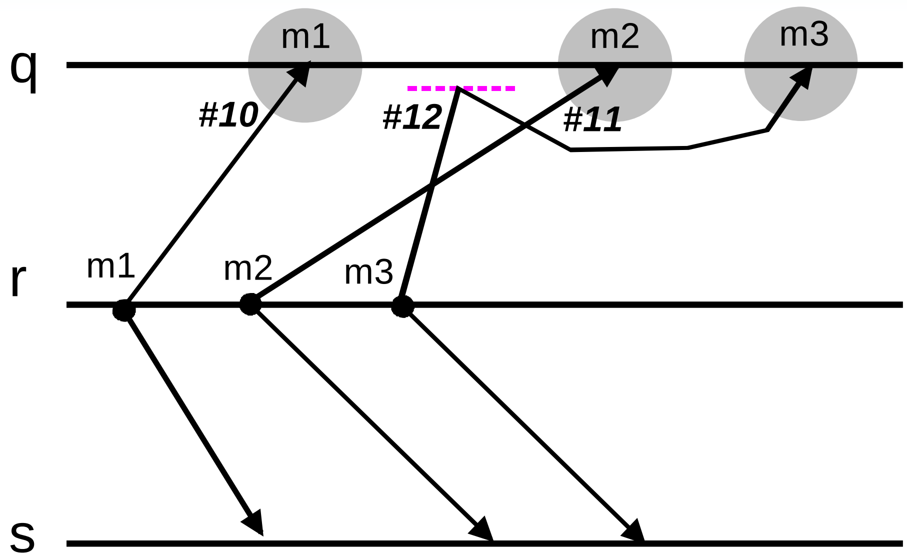
Causal Ordering implementations

When not?

- When there are interactions outside the ordering protocol, logical ordering yields ordering anomalies
- Temporal ordering
  - A message m1 is said to temporally precede message m2 iff: m1 and m2 are sent by the same or any two participants, respectively at real times t1 and t2, and (t2 - t1) > δ_t , $\delta_t \geq 0$
- Notes:
  - We call it δt-precedence (m1 $\xrightarrow{\delta_t}$ m2)
  - A temporal ordering can also yield a total ordering very easily

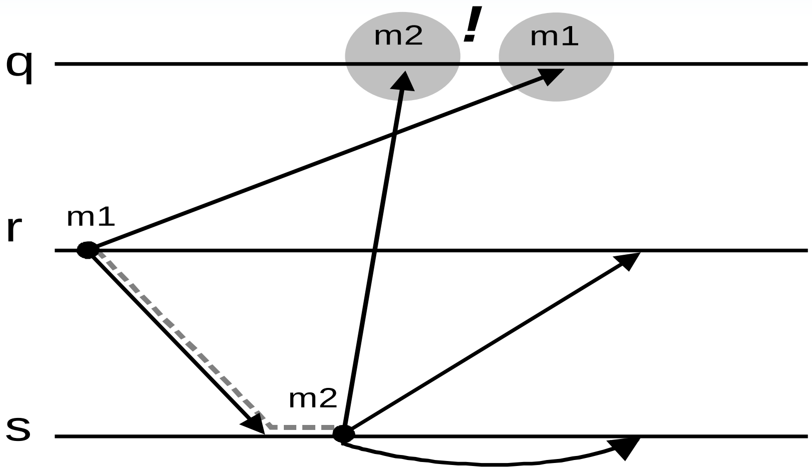
Operations in FIFO order

The most intuitive ordering



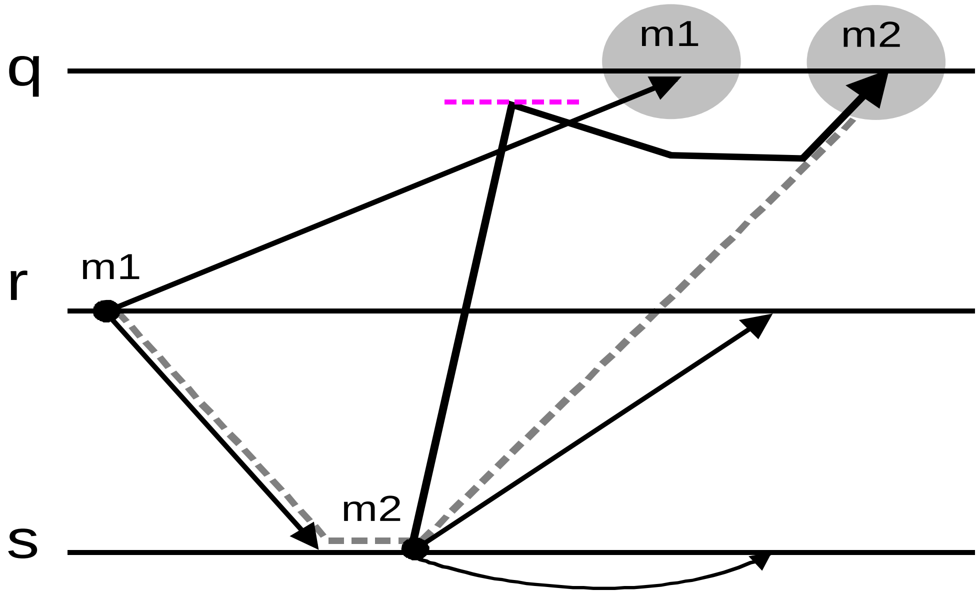
- r is solving a problem by executing 3 modules in sequence
- He disseminates intermediate results (m1, m2, m3) to s and q, who perform the second phase, which depends on the sequence order
- q got m1 with #10 and then m3 with #12, he knows m2 with #11 is missing and delays delivery of m3 until m2 arrives and only then it delivers messages m2 and m3 in that sequence
- **NB**: in complex protocols, reception often different from delivery

When FIFO order is insufficient



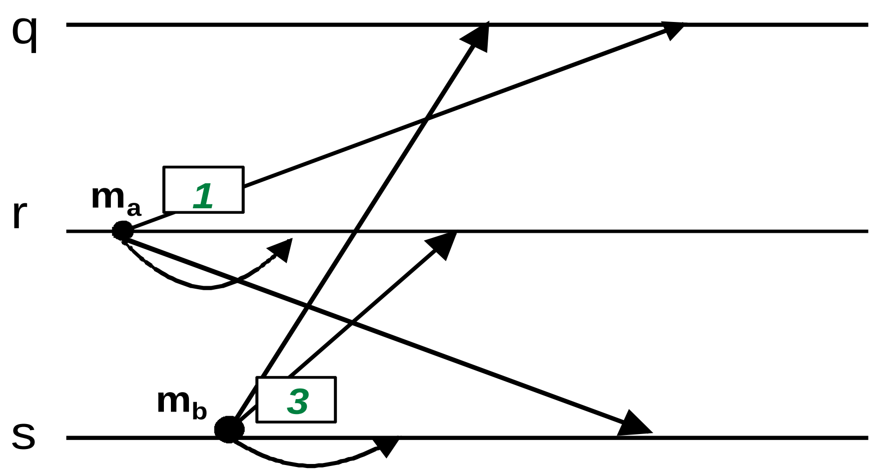
- Problem was complex, so r breaks his job in steps, asking s to perform step 2 after he does step 1, which he signals with m1
- s executes step 2 when m1 arrives, after which it send m2
- **Problem**: m1 got delayed, it will be delivered to q after m2
- Since q waits for messages in the order they were issued to perform the second phase, the application fails
- What went wrong is that FIFO protocol does not capture m1 $\rightarrow$ m2 causal relation and order inversion takes place
- Cannot be used if competing senders also exchange messages

Solution: causal order



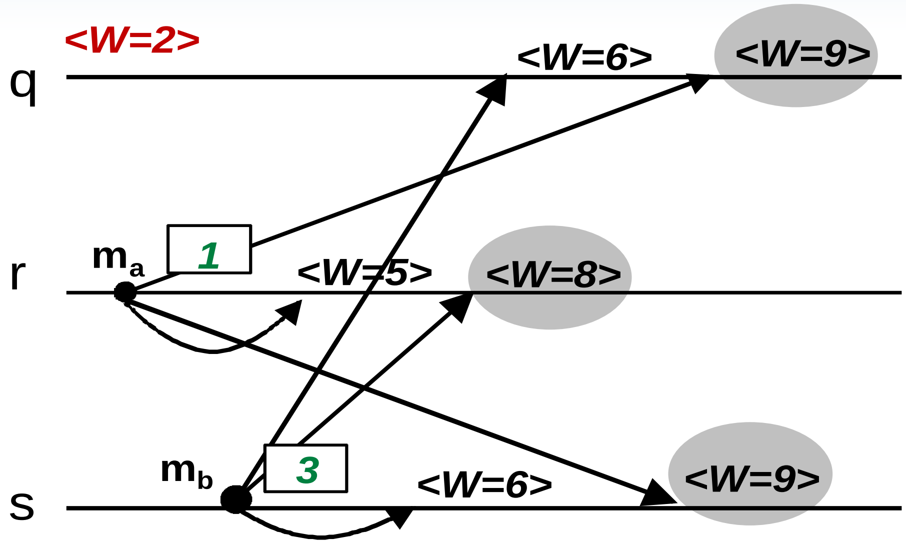
- FIFO is expanded to causal and encompasses all nodes: m1 $\rightarrow$ m2 is now recognised
- m1 is delayed to q, but q delays delivery of m2, to fulfil causal delivery

Operations in causal order



- r leads a team work performing some computations
- Result is accumulated in variable W, update function compares W previous state to new result, takes greatest and adds 3
- Errors in previous works make r request all steps done in parallel by r, q and s, and results disseminated to all, to compare results and replicate W. Any one finishing a step posts result to all including himself, in causal order
- **If everybody is doing the same steps, it is expected for W to be the same everywhere**

When causal order is insufficient

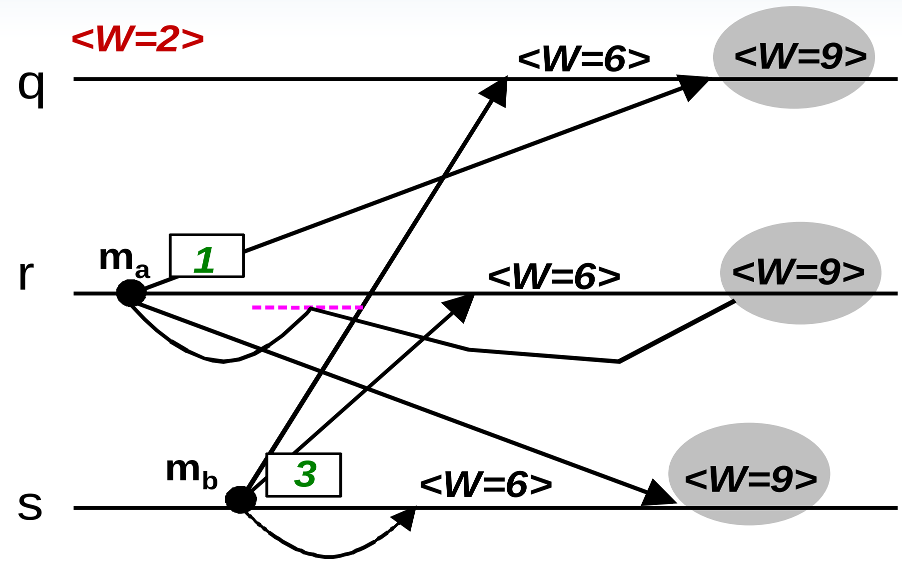


- Initially W=2, and r and s disseminate their results concurrently
- So, causal order protocol does not order them:
  - m_a = <1> is received first at r, max(2,1)=2, so W=2+3=5
  - Then m_b = <3> is received, max(5,3)=5, so W=5+3=8
  - m_b = <3> is received first at q, max(2,3)=3, so W=3+3=6
  - Then m_b = <3> is received, max(6,3)=6, so W=6+3=9
- This violates replicated computation correctness and subsequent steps depending on the value of W will not be consistent

Total ordering implementations

- **Total ordering**
  – Any two messages delivered to any pair of participants are delivered in the same order to both participants
  – **Example**: sending operation or update requests to replicas of a server, so that they execute them in the same order and produce the same result and/or assume the same state

Solution: total order



- Previous problem is solved with total order
- Active replica management requires total order, be it causal or not
- In which cases can we do active replicas without total order?
  - Think....